Title

Pedro Henrique Centenaro[*]
Supervisor: Luiz Rafael Santos

Abstract. TBD

[*]Undergraduate Control, Automation and Computation Engineering student at UFSC/Campus Blumenau.

1 Introduction

TBD

2 Knapsack problems

Informally, knapsack problems (KPs) may be described as finding the optimal combination of items from a set. Usually, this decision involves a certain value intrinsic to each item, and the objective is to maximize the total value of the selected items. To complicate this decision, items may have certain limiting values associated with them, which means not all of them may be selected.

The simplest example of a KP is the binary knapsack problem (BKP), also known as 0-1 knapsack problem [1]. It consists of a list of n items, each with a value v_i and a weight w_i associated with them, for $i \in \{1, \ldots, n\}$. A subset of items must be selected in order to maximize the total value, subject to a maximum weight capacity[1] C. One may thus formulate the BKP as Integer Program (1),

$$\begin{aligned} \max \sum_{i=1}^{n} x_i v_i & \\ \text{s. t.} \sum_{i=1}^{n} x_i w_i \leq C&, \\ x_i \in \{0,1\} \quad \forall i \in \{1, \ldots, n\}&, \end{aligned} \tag{1}$$

where x_i are decision variables that specify whether item $i \in \{1, \ldots, n\}$ has been selected or not.

By introducing slight variations to this formulation, it is possible to obtain many different uses for the KP:

- Unbounded knapsack problem (UKP): By letting $x_i \in \mathbb{N}$, the number of units of each item available is unlimited.
- Fractional knapsack problem (FKP): By letting $x_i \in [0, 1]$, a fraction of each item can be selected.
- Subset sum problem (SSP): When $v_i = w_i \, \forall i \in \{1, \ldots, n\}$, the objective can be interpreted as maximizing the weight of the knapsack. Furthermore, if we demand that $\sum_{i=1}^{n} x_i w_i = C$, then the problem becomes finding the combination of items that produces sum C exactly [7].

The KP can be further complicated by taking into account multiple knapsacks and constraints. In this section, we focus on the latter.

[1]This limiting factor, which can be interpreted as a bag or a container, is what gives the KP its name.

2.1 Simple algorithms for the binary knapsack problem

The selection of subsets implies that an upper bound on the number of solutions to the BKP is 2^n. Therefore, a natural first solution to the problem is a recursive binary tree algorithm [6]. Let $I = \{1, \ldots, n\}$ be the set of items to be put in the knapsack. Then, by representing the values and weights of items as entries in vectors $v, w \in \mathbb{R}^n$, respectively, and the remaining knapsack weight capacity as $c \in \mathbb{R}$, we arrive at Algorithm 1 for an instance $\phi(n, c, X, s)$ of the problem, where $X \subseteq I$ is the set of selected items and s is the sum of said items' values at the node.

Algorithm 1 Recursive binary tree algorithm for the KP: $\phi(n, c, X, s)$

```
 1: if $n = 0$ then
 2:     return $(X, s)$
 3: $X_1, s_1 := \phi(n-1, c, X, s)$
 4: $X_2, s_2 \leftarrow (\emptyset, \emptyset)$
 5: if $w_n \leq c$ then
 6:     $X_2, s_2 := \phi(n-1, c-w_n, X \cup \{n\}, s+v_n)$
 7: $s := max(s_1, s_2)$
 8: $X_3 \leftarrow 0$
 9: if $v = v_1$ then
10:     $X_3 := X_1$
11: else
12:     $X_3 := X_2$
13: return $(X_3, s)$
```

The algorithm consists of a top-bottom phase, in which the nodes are generated, and a bottom-top phase, in which the optimal solution is found. The top-bottom phase begins at Line 3, where a new instance of ϕ is called with item n discarded from the solution. Subsequently, if there is enough capacity left in the knapsack, a second node is generated, with n included in the solution.

When $n = 0$ (Line 1), it is impossible to generate a new node. Thus, the only remaining action is to return the solution set at the node and its value sum. Once the whole tree has been generated, Line 7 is reached. From this point on, the bottom-top phase begins, with each node selecting the best solution its children have to offer. Hence, the optimal solution is returned by the root node.

Due to the combinatorial nature of the KP, Algorithm 1 is only efficient at solving small problems [6].

Example 2.1. Suppose we have a knapsack with maximum capacity $C = 5$ and three items. The weight vector is $w = [1, 2, 3]$ and the value vector is $v = [2, 4, 3]$. By applying Algorithm 1 to this problem, we obtain the tree in Figure 1.

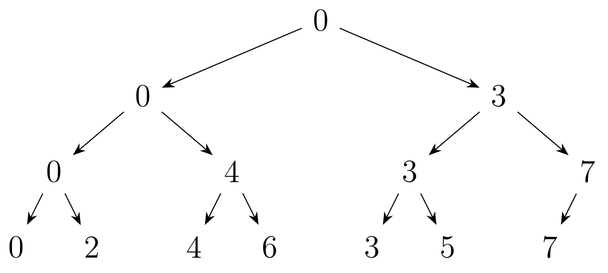

Figure 1: BKP binary tree algorithm example.

Each node in the tree represents the value sum of a feasible combination of items. It immediately becomes clear that the optimal solution in this case is 7, with the combination of items 2 and 3. However, this is not obvious to the computer, which must solve every recursion and return to the root node first.

Dynamic programming [4, 11] approaches to the KP exist [5, 6, 8]. The idea behind these approaches is to generate a table with solutions to subproblems of the KP, using previous calculations to shorten the path to the optimal solution.

Let $T \in \mathbb{R}^{(n+1)\times(c+1)}$ be the dynamic programming table for the KP. Algorithm 2 exemplifies how T can be constructed and used to arrive at the optimal KP solution.

Algorithm 2 Dynamic programming algorithm for the KP: $\Phi(n, c)$

```
1: for $i$ from 1 to $n+1$ do
2:     $T_{i,1} \coloneqq 0$
3: for $j$ from 2 to $c+1$ do
4:     $T_{1,j} \coloneqq 0$
5:     for $i$ from 2 to $n+1$ do
6:         $T_{i,j} \leftarrow T_{i-1,j}$
7:         if $w_i \leq j$ then
8:             $T_{i,j} \leftarrow \max(T_{i,j}, v_{i-1} + T_{i-1,j-w_{i-1}})$
9: $n_i, c_i, X \leftarrow (n, c, \emptyset)$
10: while $c_i > 0$ do
11:     if $T_{n_i+1,c_i+1} \neq T_{n_i,c_i+1}$ then
12:         $X \leftarrow X \cup n_i$
13:         $c_i \leftarrow c_i - w_n$
14:     $n_i \leftarrow n_i - 1$
15: return X
```

$T_{i,j}$ represents the optimal value for a subproblem of the KP with $i - 1$ items and knapsack capacity $j-1$. Consequently, no set of items can satisfy $j=1$, and so $T_{i,1} = 0\,\forall i \in \{1,\dots,n+1\}$. Furthermore, since there are no items to select when $i=1$, it follows that $T_{1,j} = 0\,\forall j \in \{1,\dots,c+1\}$. This explains lines 1 through 4.

Lines 5 through 8 fill the remaining entries in the table. If $w_{i-1} > j-1$, we simply keep the best solution found so far for capacity $j-1$. Otherwise, we compare said solution with what would be obtained if $i-1$ was selected. The larger value becomes entry $T_{i,j}$.

Once table T is finished, starting at Line 10, the process of determining what items constitute an optimal solution occurs. Following the logic of lines 5 through 8, if $T_{i,j} = T_{i-1,j}$, then the current item has not been selected and we may move on to the next item. Otherwise, the item is included in the final solution and we move on to the next item with less capacity, $T_{i-1,j-w_i}$.

Though better than the binary tree approach, the dynamic programming algorithm's complexity is exponential as well [6]. In fact, this approach is notably inefficient for large capacity values [8].

Example 2.2. Suppose the following problem suggested by Feofiloff [6]: We have 4 items and a capacity 5. The weights and values of items are, respectively, $w = [4,2,1,3]$ and $v = [500,400,300,450]$. To find the optimal solution to this BKP, we start by applying the part of Algorithm 2 to generate Table 1.

0	0	0	0	0	0
0	0	0	0	500	500
0	0	400	400	500	500
0	300	400	700	700	800
0	300	400	700	750	850

Table 1: BKP dynamic programming table example.

Next, we start looking for the solution from the table's bottom right entry. We note that $T_{5,6} \neq T_{4,6}$. This means that item $4 \in X$ (the solution set). Next, we investigate item 3. Since $4 \in X$, that means the capacity is now at $5 - w_4 = 2$. So we look at $T_{4,3}$, which is equal to $T_{3,3}$. We conclude that $3 \notin X$. Moving up a row, $T_{3,3} \neq T_{2,3}$, which means $2 \in X$. By repeating this one more time, we conclude that $X = \{2,4\}$.

Along with these algorithms, an integer programming model of the BKP was implemented in JuMP.

2.2 Models for multidimensional knapsack problems

Since the BKP contains a single capacity constraint (see Integer Program (1)), we can think of each item as a line segment whose length is w_i. It follows that the sum of lengths of the items must not exceed capacity C.

We wish to extrapolate this geometric interpretation of the BKP in a way that allows us to solve problems related to areas and volumes. In other words, we want to formulate two- and three-dimensional knapsack problems. To arrive at such formulations, first consider the multidimensional knapsack problem (MKP) [10], formulated in Integer Program (2).

$$\begin{aligned} \max & \sum_{i=1}^{n} v_i x_i \\ \text{s. t.} & \sum_{i=1}^{n} w_{ji} x_i \leq C_j, \quad \forall j \in \{1, \ldots, d\} \\ & x_i \in \{0, 1\}, \quad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{2}$$

A number of differences can be spotted in this model, compared to the BKP's. The first is the introduction of parameter $d \in \mathbb{R}$, the number of dimensions that are being taken into account. Capacity $C \in \mathbb{R}$ becomes $C \in \mathbb{R}^d$, and each item $i \in \{1, \ldots, n\}$ now has an associated vector $w_{ji} \in \mathbb{R}^d$ describing its weight across the problem's dimensions.

Specifically in the case of two dimensions, we may rewrite this formulation as Integer Program (3).

$$\begin{aligned} \max & \sum_{i=1}^{n} v_i x_i \\ \text{s. t.} & \sum_{i=1}^{n} w_i x_i \leq W, \\ & \sum_{i=1}^{n} h_i x_i \leq H, \\ & x_i \in \{0, 1\}, \quad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{3}$$

Here, w_i and h_i represent the width and height of item i, while W and H represent the width and height of the container where we wish to place the items. We assume both the items and the container to be rectangular.

Suppose we want to place three items within a 30×20 container. The width, height and value vectors for this problem are $w = [10, 10, 5]$, $h = [5, 10, 15]$ and $v = [10, 15, 5]$. Through the plot on the left in Figure 2, we can see that this is a trivial task. All items are selected and the optimal value is 30.

Our MKP formulation begs to differ: Only items 1 and 2 are selected, for an optimal value of 25. Why? The plot on the right illustrates the issue: The MKP is applying the line segment logic to multiple dimensions, as if each chosen subset of items ended up constituting a larger rectangle of dimensions $\sum_{i=1}^{n} w_i \times \sum_{i=1}^{n} h_i$.

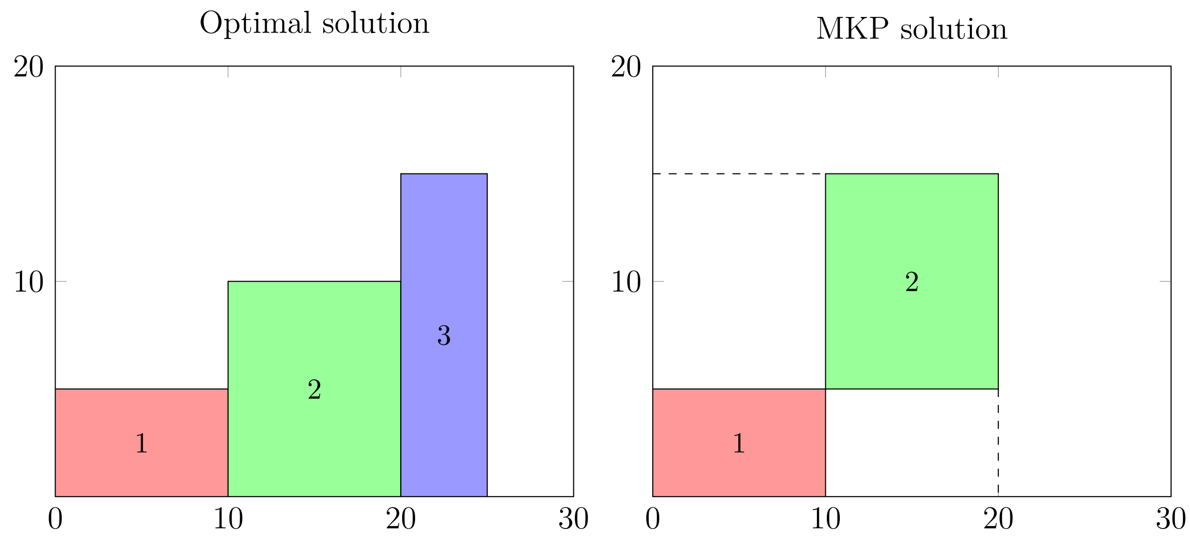

Figure 2: Bidimensional items within a 30×20 container.

We need to sophisticate our model's constraints in order to solve 2D knapsack problems. We begin by introducing variables $x_i, y_i \in \mathbb{R}$, which describe the position of each item's bottom-left corner. It immediately becomes clear that every selected item $i \in \{1, \ldots, n\}$ must obey Constraint Set (4),

$$\begin{aligned} 0 \leq x_i &\leq W, \\ 0 \leq x_i + w_i &\leq W, \\ 0 \leq y_i &\leq H, \\ 0 \leq y_i + h_i &\leq H, \end{aligned} \tag{4}$$

where W and H are the width and height of the container, respectively.

Our task is complicated by the need to avoid item overlap. To accomplish this, we adopt the approach described by Kalvelagen [9]. For every selected item $i, j \in \{1, \ldots, n\}$

with $i \neq j$, at least one inequality from Constraint Set (5) must hold true.

$$\begin{aligned} x_i + w_i &\leq x_j, \\ x_j + w_j &\leq x_i, \\ y_i + h_i &\leq y_j, \\ y_j + h_j &\leq y_i. \end{aligned} \tag{5}$$

If we force all of these constraints upon the model, that might result in slower solve times. Thus, we need a way to enforce one constraint *or* another. In other words, we have *alternative constraints* [4].

To implement this behavior in a 2DKP model, Kalvelagen [9] resorts to *conditional constraints*, defined by Bradley, Hax, and Magnanti [4] as constraints of the form

$$f_1(x_1, x_2, \ldots, x_n) > b_1 \Rightarrow f_2(x_1, x_2, \ldots, x_n) \leq b_2. \tag{6}$$

In our case, the desired relations are given by variables $\delta_{i,j,k} \in \{0, 1\}$, $k \in \{1, \ldots, 4\}$:

$$\delta_{i,j,1} = 1 \Rightarrow x_i + w_i \leq x_j, \tag{7}$$
$$\delta_{i,j,2} = 1 \Rightarrow x_j + w_j \leq x_i, \tag{8}$$
$$\delta_{i,j,3} = 1 \Rightarrow y_i + h_i \leq y_j, \tag{9}$$
$$\delta_{i,j,4} = 1 \Rightarrow y_j + h_j \leq y_i. \tag{10}$$

These constraints can be mathematically implemented as *big M constraints*[2]. The idea is to introduce a constant $M \to +\infty$ and rewrite the constraints as

$$x_i + w_i - M(1 - \delta_{i,j,1}) \leq x_j, \tag{11}$$
$$x_j + w_j - M(1 - \delta_{i,j,2}) \leq x_i, \tag{12}$$
$$y_i + h_i - M(1 - \delta_{i,j,3}) \leq y_j, \tag{13}$$
$$y_j + h_j - M(1 - \delta_{i,j,4}) \leq y_i. \tag{14}$$

Finally, we add the following constraint:

[2]the concept is similar to the big M method for solving linear programming problems [2, 3]

$$\sum_{k=1}^{n} \delta_{i,j,k} \geq 1. \tag{15}$$

For $M \to +\infty$, if a certain constraint is not selected, it's obvious that the inequations hold. This remains true for any item smaller than the container when $M = \max\{W, H\}$.

It is possible to introduce rotations to this model. For this purpose, we create new variables σ_i. When item i is rotated, $\sigma_i = 1$. Otherwise, $\sigma_i = 0$. Then, the following constraints must be introduced:

$$x'_i = x_i + (1 - \sigma_i)w_i + \sigma_i h_i, \tag{16}$$

$$y'_i = y_i + (1 - \sigma_i)h_i + \sigma_i w_i. \tag{17}$$

Having made all of these considerations, we can finally produce our continuous 2DKP model. Here, s_i are the decision variables. For brevity, consider $N = \{1, \ldots, n\}$.

$$\max \sum_{i \in N} v_i s_i \tag{18a}$$

$$\text{s. t.} 0 \leq x_i \leq W, \quad \forall i \in N, \tag{18b}$$

$$0 \leq x'_i \leq W, \quad \forall i \in N, \tag{18c}$$

$$0 \leq y_i \leq H, \quad \forall i \in N, \tag{18d}$$

$$0 \leq y'_i \leq H, \quad \forall i \in N, \tag{18e}$$

$$x'_i = x_i + (1 - \sigma_i)w_i + \sigma_i h_i, \quad \forall i \in N, \tag{18f}$$

$$y'_i = y_i + \sigma_i w_i + (1 - \sigma_i)h_i, \quad \forall i \in N, \tag{18g}$$

$$x'_i - M(3 - \delta_{i,j,1} - s_i - s_j) \leq x_j, \quad \forall i \in N, \forall j \in N, i < j, \tag{18h}$$

$$x'_j - M(3 - \delta_{i,j,2} - s_i - s_j) \leq x_i, \quad \forall i \in N, \forall j \in N, i < j, \tag{18i}$$

$$y'_i - M(3 - \delta_{i,j,3} - s_i - s_j) \leq y_j, \quad \forall i \in N, \forall j \in N, i < j, \tag{18j}$$

$$y'_j - M(3 - \delta_{i,j,4} - s_i - s_j) \leq y_i, \quad \forall i \in N, \forall j \in N, i < j, \tag{18k}$$

$$\sum_{k=1}^{4} \delta_{i,j,k} \geq 1, \quad \forall i \in N, j \in N, \tag{18l}$$

$$\delta_{i,j,k} \in \{0, 1\}, \quad \forall i \in N, j \in N, k \in \{1, 2, 3, 4\} \tag{18m}$$

$$\sigma_i \in \{0, 1\}, \quad \forall i \in N, \tag{18n}$$

$$s_i \in \{0, 1\}, \quad \forall i \in N. \tag{18o}$$

Note that we have made a few changes to Constraints (18h) to (18k). To make them even stronger, we require both items to have been selected for the constraints to take effect. Moreover, since these constraints go both ways, it is only necessary to evaluate them for $i < j$.

We have written an implementation of the model proposed by Kalvelagen [9], which is an equivalent to Mixed-Integer Program (18) that is more geared towards computational work, making heavy use of matrices for conciseness.

Kalvelagen [9] also made a discrete 2DKP model, referred to as *covering model.* Say each item's width and height is represented by continuous variables $\tilde{w}_i$ and $\tilde{h}_i$. We begin by discretizing these quantities as $w_i = \lceil \tilde{w}_i \rceil$ and $h_i = \lceil \tilde{h}_i \rceil$. The container is discretized with $W = \lfloor \tilde{W} \rfloor$ and $H = \lfloor \tilde{H} \rfloor$.

This initial process allows for the interpretation of items as collections of cells. As shown in Figure 3, a 7.23×5.55 item (darker blue) takes 8×6 cells (lighter blue).

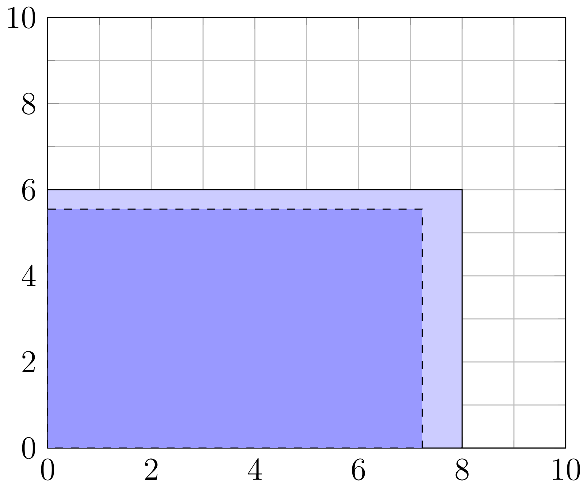

Figure 3: Discretization of an item for the 2DKP covering model.

In the covering model, we introduce binary matrix $A \in \mathbb{R}^{n \times 2 \times W \times H}$. If item k can be placed at cell (i, j) with rotation r without leaving the boundaries of the container, then $A_{k,i,j,r} = 1$. Otherwise, $A_{k,r,i,j} = 0$. By convention,

$$r = 1 \Rightarrow \text{item is not rotated,}$$
$$r = 2 \Rightarrow \text{item is rotated.}$$

Next, we introduce binary matrix $C \in \mathbb{R}^{n \times 2 \times W \times H \times W \times H}$. If cells (i', j') are covered as a result of item k being placed at cell (i, j) with rotation r, then $C_{k,r,i,j,i',j'} = 1$. Otherwise,

$C_{k,r,i,j,i',j'} = 0$.

Lastly, we make the decision variables $s_{k,r,i,j}$, since item placement is based on the top-left cell of each item. This is enough to formulate Integer Program (19).

$$\max \sum_{n,r,i,j|A_{n,r,i,j}=1} v_i s_{k,r,i,j} \tag{19a}$$

$$\text{s. t.} \sum_{k,r,i,j|C_{k,r,i,j,i',j'}=1} s_{k,r,i,j} \leq 1, \quad \forall i' \in \{1,\dots,W\},\ j' \in \{1,\dots,H\}, \tag{19b}$$

$$\sum_{r,i,j|A_{k,r,i,j}=1} s_{k,r,i,j} \leq 1, \quad \forall k \in \{1,\dots,n\} \tag{19c}$$

$$s_{k,r,i,j} \in \{0,1\}. \tag{19d}$$

We have also written an implementation of this model. The model we implemented is the one presented by Kalvelagen [9], which is more memory-efficient than Integer Program (19), since it groups items of the same type together in the matrices.

Example 2.3.

References

[1] Maram Assi and Ramzi A Haraty. “A survey of the knapsack problem.” In: *2018 International Arab Conference on Information Technology (ACIT)*. IEEE. 2018, pp. 1–6.

[2] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows*. 4th ed. Hoboken, New Jersey: Wiley, 2010.

[3] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. 4th ed. Belmont, Massachusetts: Athena Scientific and Dynamic Ideas, 1997.

[4] Stephen P Bradley, Arnoldo C Hax, and Thomas L Magnanti. “Applied mathematical programming.” In: *(No Title)* (1977).

[5] Erik Demaine and Srini Devadas. *Polynomial Time vs Pseudo-Polynomial Time*. 2011. URL: https://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf (visited on 02/09/2024).

[6] Paulo Feofiloff. *Mochila booleana*. 2020. URL: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool (visited on 02/09/2024).

[7] Paulo Feofiloff. *Subset sum*. 2020. URL: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-subsetsum (visited on 02/09/2024).

[8] Maya Hristakeva and Dipti Shrestha. “Different approaches to solve the 0/1 knapsack problem.” In: *The Midwest Instruction and Computing Symposium*. 2005.

[9] Erwin Kalvelagen. *2d knapsack problem*. 2021. URL: http://yetanothermathprogrammingconsult blogspot.com/2021/10/2d-knapsack-problem.html (visited on 02/10/2024).

[10] Soukaina Laabadi, Mohamed Naimi, Hassan El Amri, Boujemâa Achchab, et al. “The 0/1 multidimensional knapsack problem and its variants: A survey of practical models and heuristic approaches.” In: *American Journal of Operations Research* 8.05 (2018), p. 395.

[11] David B Wagner. “Dynamic programming.” In: *The Mathematica Journal* 5.4 (1995), pp. 42–51.