Decision Making with Business Analytics - Project
The Knapsack Problem

by

Matthijs Kroesen - 2060519
Jarno Ringhs - 2039120

Tilburg School of Economics and Management
Tilburg University

Date: October 11, 2024

Contents

Chapter 1

The Knapsack Problem

The Knapsack Problem is a well-known combinatorial optimization problem [2], which is considered as one of the 'easier' $\mathcal{NP}$-hard problems [2]. The Knapsack Problem can be intuitively interpreted as filling a knapsack with the most valuable items possible such that the chosen items fit in the knapsack. Given a non-negative integer amount of items n, item values $v_1, \ldots, v_n$, item weights $w_1, \ldots, w_n$, and total capacity W, the Knapsack Problem can be modeled as the following linear optimization problem [4]:

$$x_i = \begin{cases} 1 & \text{if if item } i \text{ is chosen in the knapsack} \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0,1\} \qquad \forall i \in \{1,2,\ldots,n\} \end{aligned}$$

1.1 Binary Programming (BP)

The solver Gurobi is employed in Python to solve the Knapsack problem using Binary Programming. Binary Programming is ordinary Integer Linear Programming, for which the decision variable x_i is not only integer, but also binary (i.e. x_i can only take value 0 or 1. Firstly, the Gurobi packaged is imported, and the indices $i \in \{1,2,\ldots,n\} = N$ are defined. Then, the decision variables x_i are defined for every $i \in N$, and dictionaries are created in which w_i and v_i are stored for every $i \in N$. Following, the objective function $\max_x \sum i = 1^n v_i x_i$ and the constraint $\sum_{i=1}^{n} w_i x_i \leq W$ are defined in the Gurobi framework. Finally, the Knapsack Problem is optimized using the Gurobi software, and we export the objective values. The Python code for this can be found in the file `Standard_methods.py` under the function `BinaryProgrammingKnapsack`(n, W, v, w). The optimal values and running times for several generated Knapsack instances that Binary Programming solved can be found in Appendix A.2.

1.2 Dynamic Programming (DP)

When using Dynamic Programming, one solves a combinatorial optimization problem by using a recursive relationship between smaller subproblems of the problem. For the Knapsack Problem, define the following recursive optimization rule:

$$OPT(i,w) = \begin{cases} 0 & \text{if if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

The pseudo-code for the Dynamic Programming algorithm implementing this recursive relationship between instances for the Knapsack Problem, as presented by Dobre [1], is implemented in Python, for which the code can be found in the file `Standard_methods.py` under the function `DynamicProgrammingKnapsack`(n, W, v, w). The pseudocode can be found in Appendix A.1 as Algorithm 1. The optimal values and running times for several generated Knapsack instances that Dynamic Programming solved can be found in Appendix A.2.

1.3 Greedy Heuristic (GH)

The basic premise of the Greedy Heuristic for the Knapsack Problem is to sort the items descending according the ratio $\frac{v_i}{w_i}$, and to repeatedly add the item with the highest ratio that does not exceed the capacity constraint to the knapsack. The pseudocode for this greedy heuristic for the Knapsack Problem, as presented by Dobre (2024), can be found in Appendix A.1 as Algorithm 2. This pseudocode is also implemented in Python, for which the code can be found in the file `Standard_methods.py` under the function `GreedyHeuristicKnapsack`(n, W, v, w). The optimal values and running times for several generated Knapsack instances that this Greedy Heuristic solved can be found in Appendix A.2.

1.4 Neural Dynamic Programming (NDP)

The main problem with the DP algorithm is the size of the table, which is in particular an issue when W becomes exponentially big. Therefore, we came up with a neural network-based Q-learning algorithm where the runtime of the algorithm does not depend on W. The central idea behind our algorithm is that a Neural Network (NN) is used to determine whether an item should be added to the knapsack or not, instead of a (very large for large W) Dynamic Programming table. An NN is iteratively trained using Q-learning, using $K = 10$ 'episodes' (i.e. iterations), where the NN estimates the values of the underlying Q-table. The pseudocode for this Neural Dynamic Programming algorithm can be found in Appendix A.1 as Algorithm 3. The actual Python code can be found in the file `knapsackSolver.py` under the function `knapsackSolver`(n, W, v, w).

For the Q-learning, the action space is defined as $\mathcal{A} = \{0, 1\}$ (include the item (1) in the knapsack or not (0)). Denote the weight left in the knapsack as w_{left}. Then the state $\mathcal{S}_i$ is defined as $\mathcal{S}_i = [v_i, w_i, \frac{v_i}{w_i}, w_{\text{left}}]$, $\forall i \in \{1, 2, \ldots, n\}$. The parameters are set to balance exploitation of the best solutions versus exploration of new solutions in the solution space: namely $\epsilon = 0.9$, the decay factor $\beta = 0.995$, and the discount factor $\gamma = 0.6$. Firstly, the NN is initialized. The NN has as input a state S_i, has a hidden layer with 20 nodes, and has as 2 output nodes. Each output note represents the expected gain of an action. Thus, the NN represents the choice which action $a \in \mathcal{A}$ we should take given state $\mathcal{S}_i \in \mathcal{S}$. The hidden layer has a Rectified Linear Unit activation function and the output layer has a linear activation, as the NN is predicting values that are not necessarily on the interval [0,1]. The action $\text{argmax}(NN(\mathcal{S}_i) = a^* \in \mathcal{A}$ is defined as the best action to take. Finally, the NN is compiled utilizing a mean squared error loss function and the Adam optimizer (an improved Stochastic Gradient Descent method).

Then, we start the actual Q-learning algorithm. We perform this algorithm $K = 10$ times. Each iteration $\kappa \in \{1, 2, \ldots, k\}$ starts with an ϵ-greedy policy. In such a policy, we take a random number $z \sim UNIF(0, 1)$, and if $z \geq \epsilon$, then the NN predicts for input $\mathcal{S}_i \in \mathcal{S}$ which action $a \in \mathcal{A}$ has the highest expected gain (i.e. whether we should put item i in the knapsack or not). For each iteration, We start at item 1 and loop through all items available. We first

define the state $\mathcal{S}_i$, which is the input for the NN. Thus, the NN predicts which action acquires the highest expected gain for some state. If $\epsilon > z$, then a is chosen as a random integer on $[0, 1]$ to encourage exploration of new solutions. Ultimately, with a probability of $\epsilon \in (0, 1)$, a random action is chosen, and with a probability of $1 - \epsilon$ the action with maximal expected gain is chosen. This ϵ-greedy strategy is used to stimulate exploration in earlier stages of the algorithm. If an item is added to the knapsack and $w_i < w_{\text{left}}$, the reward is equal to the profit of the item. If it gets added and does not fit in the knapsack ($w_i > w_{\text{left}}$), the reward is equal to $-10v_i$, to punish exceeding the capacity constraint of the knapsack. If an item does not get added, the reward is 0. Furthermore, we set a target value τ for the NN. This value is equal to the reward plus the discounted gain of the optimal action in the next state, which is predicted by the NN with input $\mathcal{S}_{i+1}$ (except when $i = n$, since it is the last item considered). Now, the Q-values are computed by predicting these using the NN with $\mathcal{S}_i$ as input. The computed Q-value for the previously chose action a is replaced by the target value (we want to fit the NN on some target, to improve it in the future). With $\mathcal{S}_i$ as input the NN is fitted on these values. Then we update w_{left}. The weight w_i is subtracted from it if the item has been added and remains the same otherwise. Now we move on to a new item. For each iteration ϵ will decay by multiplying it with β, which is smaller than 1. This epsilon-decaying strategy stimulates exploitation of good solutions and makes sure exploitation is preferred at the end. However, to also have a sufficient probability of exploration at the later stages of the algorithm we have $\epsilon = \max(\epsilon \cdot \beta, 0.1)$. Ultimately, the episode with the best cumulative reward will be chosen as having generated the optimal solution, and for this solution $x_{opt} = \{0, 1\}^n$ the value $v_{opt} = x^T v$ is determined.

1.5 Generating Data

The instances used for our project are based on one of the instances proposed by Pisinger (2005b). We make use of the strongly correlated instances. The maximum weight of an item is denoted by w_{max}. For each item i of the instance, the weight w_i is drawn from a discrete uniform distribution on the interval $[1, w_{\max}]$. The corresponding profit is generated as $p_i = w_i + \lceil \frac{w_{\max}}{10} \rceil$. In contrast to [3], we make sure the profit is always an integer by taking the ceil of the division, since the instances will be also solved by dynamic programming. The capacity of the knapsack is denoted by W and defined as $\alpha \sum_{i=1}^{n} w_i$. For our instances, $\alpha = 0.4$. Ordering the items in descending order based on the profit-to-weight ratio effectively results in an ordering of the item weights. Within a small range of items, the variation in weights is minimal, making it challenging to meet the capacity constraint [3]. Therefore, the greedy heuristic should come up with solutions that are most of the cases non-optimal. Nonetheless, looking at Table A.1 we conclude that the greedy heuristic is most of the times close to the optimal solution. In addition, we see that the running time of DP increases in a non-linear fashion when w_{max} and therefore also the capacitity W is increased.

1.6 Comparison

Table A.1 shows the results attained by applying the algorithms to our generated instances. The runtime of the NDP algorithm stays relatively constant when n is kept constant and w_{max} is increased. The runtime of DP grows in a non-linear way and from $w_{max} \leq 10^5$ on NDP is faster than DP. GH performs relatively well on the instances generated, since it is always close to the optimal value. Nonetheless, the NDP model is able to beat GH in 6 out of 18 cases. In general we see that NDP beats GH on the smallest instances with $n = 10$.

Bibliography

[1] Dobre, C. (2024). Decision making for business analytics: Formulating dynamic programming problems (lecture 7).

[2] Pisinger, D. (2005a). Where are the hard knapsack problems?

[3] Pisinger, D. (2005b). Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284.

[4] Sotirov, R. and R., P. (2023). Combinatorial optimization: The knapsack problem (lecture 6).

Appendix A

Appendix

A.1 Pseudocodes

Pseudocode for the Dynamic Programming for the Knapsack problem of Section 1.2:

Algorithm 1 Dynamic Programming for the Knapsack Problem

Input: $n, W, v_1, \ldots, v_n, w_1, \ldots, w_n$
for $w \in \{0, 1, 2, \ldots, W\}$ **do**
    $M[0, w] = 0$
end for
for $i \in \{0, 1, 2, \ldots, n\}$ **do**
    for $w \in \{1, 2, \ldots, W\}$ **do**
        if $W_i > w$ **then**
            $M[i, w] = M[i-1, w]$
        else
            $M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w-w_i]\}$
        end if
    end for
end for
Backtrack to obtain the indices I_{dyn} of optimal solution
Output: I_{dyn}

Pseudocode for the Greedy Heuristic for the Knapsack Problem of Section 1.3:

Algorithm 2 Greedy Heuristic for the Knapsack Problem

Input: $n, W, v_1, \ldots, v_n, w_1, \ldots, w_n$
Let ratio $r_i = \frac{v_i}{w_i}$, $\forall i \in \{1, 2, \ldots, n\}$
Make matrix $M = [r; v; w]$ and Sort M descending on row r
Let $w_k = 0, v_k = 0, i = 1$
while $w_k < W$ and $i < n$ **do**
    if $w_k + M_{2,i} < W$ **then**
        Let $w_c \leftarrow w_c + M_{2i}$, $v_c \leftarrow v_c + M_{3i}$, $I_{greedy} \leftarrow I_{greedy} \cup i$, and $i \leftarrow i + 1$
    end if
end while
Output: I_{greedy}

Pseudocode for Neural Dynamic Programming for the Knapsack Problem of Section 1.4:

Algorithm 3 Neural Dynamic Programming for the Knapsack Problem

Input: $n, W, v_1, \ldots, v_n, w_1, \ldots, w_n, K = 10$
Let $\epsilon = 0.9, \beta = 0.995, \gamma = 0.6, z_{opt} = 0$
Let $\mathcal{S}_i = [w_i, v_i, \frac{v_i}{w_i}, w_{left}]$, $\forall i \in \{1, 2, \ldots, n\}$
Initialize Neural Network $NN(x)$**:**
    Input $x = \mathcal{S}_t$
    Hidden Layer of 20 nodes ▷ Rectified Linear Unit activation function.
    Output $\mathbf{y} = [y_0, y_1]$ ▷ Linear activation function.
                ▷ y_a is expected gain of selecting item i ($a = 1$) or not ($a = 0$).
for $\kappa = 1, \ldots, K$ **do**
    Let $z_\kappa = 0$ and $x_{\kappa,i} = 0$, $\forall i \in \{1, 2, \ldots, n\}$
    for $i = 1, \ldots, n$ **do**
        Let $w_{left} = W$ and $\mathcal{S}_i = [w_i, v_i, \frac{v_i}{w_i}, w_{left}]$
        Draw a random number $z \sim UNIF[0, 1]$
        if $z < \epsilon$ **then**
            Let a be a random integer on $[0, 1]$
        else
            $a = \text{argmax}(NN(\mathcal{S}_i))$
        end if
        if $a = 1$ **then**
            $w_{left} \leftarrow w_{left} - w_i$ and $x_{\kappa,i} \leftarrow 1$
            if $w_{left} < 0$ **then**
                $r \leftarrow -10 \cdot v_i$
            else
                $r \leftarrow v_i, z_\kappa \leftarrow z_\kappa + r$
            end if
        else
            $r \leftarrow 0$
        end if
        if $i < n - 1$ **then**
            $\mathcal{S}_{i+1} = [w_{i+1}, v_{i+1}, \frac{v_{i+1}}{w_{i+1}}, w_{left}]$
            Target $\tau = r + \gamma \cdot \max(NN(\mathcal{S}_{i+1}))$
        else
            $\tau = r$
        end if
        Set Q-values $Q = NN(\mathcal{S}_i)$, set $Q_a \leftarrow \tau$
        Fit $NN(\mathcal{S}_i)$ on q
        $\epsilon \leftarrow \max\{0.1, \epsilon \cdot \beta\}$,
    end for
    if $z_{opt} < z_\kappa$ **then**
        $z_{opt} \leftarrow z_\kappa$, $i_{opt} \leftarrow i$ x_{opt}, and $x_{opt} \leftarrow x_\kappa$
    end if
end for
$v_{opt} = x_{i_{opt}}^T v_{i_{opt}}$
Output: v_{opt}

A.2 Addendum: Results for different instances and solving methods of the Knapsack Problem

w_{max}	Method	n, Optimal Value, and Runtime in seconds					
		$n = 10$		$n = 30$		$n = 50$	
		Runtime	Optimal Value	Runtime	Optimal Value	Runtime	Optimal Value
10	BP	0.0156	21	0.0157	85	0.1718	140
	DP	0	21	0	85	0.0156	140
	GH	0	20	0	82	0	134
	NDP	16.0055	20	46.4432	80	78.5731	134
10^3	BP	0.0156	2560	0.0156	7944	0.7498	13426
	DP	0.0156	2560	0.1718	7944	0.4999	13426
	GH	0	2211	0	7802	0	13263
	NDP	15.6246	2393	46.978	7323	76.879	12180
10^4	BP	0.0155	24880	0.0312	83737	0.0312	126540
	DP	0.1719	24880	2.6997	83737	5.6417	126540
	GH	0	24012	0	77057	0	123428
	NDP	15.502	24404	49.2176	77738	77.2168	113647
10^5	BP	0.0326	240823	0.0326	805760	0.0468	1409874
	DP	1.8301	240823	25.0091	805760	82.582	1409874
	GH	0	240384	0	784553	0	1356180
	NDP	15.4222	223424	46.9005	754144	76.6197	1333108
10^6	BP	0.0312	2884352	0.0312	8348002	0.0312	12991439
	DP	31.615	2884352	379.3243	8348002	-	-
	GH	0	2523639	0	7927349	0	12336409
	NDP	15.3153	2728806	47.4548	7907529	79.2344	11868299
10^7	BP	0.11	31257843	0.015	87751451	0.047	131894303
	DP	-	-	-	-	-	-
	GH	0	27062699	0	80488510	0	127072347
	NDP	16.46	28472036	46.72	82282509	76.64	122980582

Table A.1: Results for different (generated) instances of and solving methods for the Knapsack Problem

No optimal values for $w_{max} > 10^7$ were computed since when performing the dynamic programming algorithm for the Knapsack Problem caused our laptops and campus PCs to crash due to memory errors. Hence, no comparisons could be made to dynamic programming results, thus generated instances with $w_{max} > 10^7$ were omitted. When an entry in Table A.1 is marked with a bar (-), such a memory error was encountered for dynamic programming. The results are generated using NumPy random seed 6.