LINEAR & QUADRATIC KNAPSACK OPTIMISATION PROBLEM

Supervisor: Prof. Montaz Ali

Alex Alochukwu, Ben Levin, Krupa Prag, Micheal Olusanya, Shane Josias, Sakirudeen Abdulsalaam, Vincent Langat

January 13, 2018

GROUP 4: Graduate Modelling Camp - MISG 2018
· The Knapsack Problem is considered to be a **combinatorial optimization problem**.

· The **best selection and configuration** of a collection of objects adhering to some objective function defines combinatorial optimization problems.
Problem Description: Knapsack Problem

To determine the number of items to include in a collection such that the total weight is less than or equal to a given limit and the total value is maximised.
Problem Description: Knapsack Problem
Mathematical Formulation: Linear Knapsack Problem

Given n-tuples of positive numbers \((v_1, v_2, ..., v_n)\), \((w_1, w_2, ..., w_n)\) and \(W > 0\). The aim is to determine the subset \(S\) of items each with values \(v_i\) and \(w_i\) that

Maximize \(\sum_{i=1}^{n} v_i x_i\), \(x_i \in \{0, 1\}\), \(x_i\) is the decision variable \hspace{1cm} (1)

Subject to: \(\sum_{i=1}^{n} w_i x_i < W\), \hspace{1cm} (2)

where \(W < \sum_{i=1}^{n} w_i\).
Quadratic Knapsack Problem

- Extension of the linear Knapsack problem.
- Additional term in the objective function that describes extra profit gained from choosing a particular combination of items.
Mathematical Formulation

Maximize \[ \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j \] \hspace{1cm} (3)

Subject to: \[ \sum_{j=1}^{n} w_j x_j < W, \quad j = \{1, 2, \ldots, n\}, \] \hspace{1cm} (4)

where \( x \in \{0, 1\} \), \( \text{Max } w_j \leq W < \sum_{j=1}^{n} w_j \).
A Knapsack model serves as an abstract model with broad spectrum applications such as:

- Resource allocation problems
- Portfolio optimization
- Cargo-loading problems
- Cutting stock problems
· The **abstract measurement** of the rate of growth in the required resources as the input $n$ increases, is how we distinguish among the complexity classes.
The following solution schemes were proposed for solving the Linear and Quadratic Knapsack Problem:

- Greedy Algorithm
- Polynomial Time Approximation Scheme
- Exact Method (Branch and Bound Algorithm)
- Dynamic Programming (Bottom - up)
1. Identify the available items with their weights and values and take note of the maximum capacity of the bag.
2. Use of a score or efficiency function, i.e. the profit to weight ratio: 
   \[ \frac{v_i}{w_i} \].
3. Sort the items non-increasingly according to the efficiency function.
4. Add into knapsack the items with the highest score, taking note of their accumulative weights until no item can be added.
5. Return the set of items that satisfies the weight limit and yields maximum profit.
Greedy algorithm for QKP

1. Sample $k$ items from the set of $n$ items.
2. Obtain a set of all pairs from the $k$ items.
3. Sort the items non-increasingly according to the efficiency function
   \[ S = \frac{d_{ij}}{w_i + w_j}. \]
4. Add into knapsack the pair of items with the highest score, ensuring that the accumulated weight does not exceed the maximum capacity.
5. Repeat steps 1 through 4 until pairs can no longer be added.
6. Fill remaining capacity with singleton items, using the previous greedy approach.
1. Consider all sets of up to at most $k$ items

$$\mathcal{F} = \{F \subset \{1, 2, \ldots, n\} : |F| \leq k, w(F) < W\}$$

2. For all $F$ in $\mathcal{F}$
   - Pack $F$ into the knapsack
   - Greedily fill the remaining capacity
   - End

3. Return highest valued item combination set
Branch and Bound Method

Branch and Bound performs systematic enumeration of candidate solutions by means of state search space.

\[ W = 10 \]

Red nodes are the nodes that are ignored because of infeasible solution through them.

Purple nodes are nodes ignored because the best solution through them is worse than current best.
Dynamic Programming (DP)

- DP: What is the idea?
- Pros?
- Cons?
Dynamic Programming: Bottom-up

1. Construct $V \in \mathbb{R}^{n \times W}$
   - $n =$ Total number of objects to be packed
   - $W =$ maximum weight capacity.
   - For $1 \leq i \leq n$, and $0 \leq w \leq W$, $V(i, w)$ stores the maximum value of variables \{1, 2, \ldots, $i$\} of size at most $w$.
2. $V(n, W)$ is the optimal value of the problem.
3. Recursion
   The process is as follows:
   
   Initialization:
   $V(0, w) = 0 \forall w \in [0, W]$ (no item); $V(i, w) = -\infty$ if $w < 0$
   
   Recursive step:
   $V(i, w) = \max(V(i - 1, w), v_i + V(i - 1, w - w_i))$ for $1 \leq i \leq n, 0 \leq w \leq W$.
   $v_i \in \bar{V}$ is the set of values of the objects to be packed while $w_i \in \bar{W}$ is their corresponding weights.
Let $W = 10$ and

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i$</td>
<td>10</td>
<td>40</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>$w_i$</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$w$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

The optimal value is $V(4, 10) = 90$. The items that give the maximum value are 2 and 4.
# Results and Discussions

**Table: Algorithm Optimality**

<table>
<thead>
<tr>
<th></th>
<th>Greedy</th>
<th>PTAS</th>
<th>Dynamic Programming</th>
<th>BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 30$</td>
<td>261</td>
<td>261</td>
<td>261</td>
<td>261</td>
</tr>
<tr>
<td>$N = 50$</td>
<td>480</td>
<td>481</td>
<td>481</td>
<td>481</td>
</tr>
<tr>
<td>$N = 100$</td>
<td>891</td>
<td>891</td>
<td>891</td>
<td>891</td>
</tr>
</tbody>
</table>
**Table: QKP optimality**

<table>
<thead>
<tr>
<th></th>
<th>$K = 0.5 \times N$</th>
<th>$K = N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 10$</td>
<td>195</td>
<td>179.5</td>
</tr>
<tr>
<td>$N = 20$</td>
<td>735.5</td>
<td>853.5</td>
</tr>
<tr>
<td>$N = 30$</td>
<td>1739.5</td>
<td>2572.5</td>
</tr>
<tr>
<td>$N = 50$</td>
<td>4755</td>
<td>7528</td>
</tr>
<tr>
<td>$N = 100$</td>
<td>18551.5</td>
<td>16855</td>
</tr>
</tbody>
</table>
1. Greedy Algorithm
   - With sorting: $\mathcal{O}(n \log n)$
   - Without sorting: $\mathcal{O}(n^2)$

2. Polynomial Time Approximation Scheme
   - $\mathcal{O}(kn^{k+1})$

3. Dynamic Programming
   - $\mathcal{O}(nW)$
Greedy Runtime

![Graph showing runtime vs problem size](image-url)
PTAS Runtime

The graph shows the runtime (in milliseconds) as a function of $k$ for different values of $n$. The x-axis represents $k$, ranging from 1 to 8. The y-axis represents time, with a logarithmic scale ranging from $10^{-2}$ to $10^3$.

- The blue line represents $n = 10$.
- The red line represents $n = 20$.
- The orange line represents $n = 30$.

As $k$ increases, the runtime for all values of $n$ grows, but the rate of growth varies depending on the value of $n$. The runtime for $n = 10$ is the lowest, followed by $n = 20$, and then $n = 30$.
DP Runtime

![Graph showing the relationship between problem size (n) and run time. The run time increases significantly as the problem size grows.]
1. Combinatorial problems are hard to solve.
3. Interesting research questions.
4. Better data.
K. Lai.  
*The Knapsack Problem and Fully Polynomial Time Approximation Schemes (FPTAS).*  

M. Ali.  
*Discrete Optimisation.*  
Lecture notes.