A Study on the 0-1 Knapsack Problem

Yifan Liu
School of Public Policy
Georgia Institute of Technology
yliu3494@gatech.edu

Qidian Gao
Department of Computer Science and Engineering
Georgia Institute of Technology
qgao67@gatech.edu

Andi Xia
Department of Computer Science and Engineering
Georgia Institute of Technology
axia35@gatech.edu

Mi Zhou
Department of Electrical and Computer Engineering
Georgia Institute of Technology
mzhou91@gatech.edu

ABSTRACT

The Knapsack problem is a well-known NP-complete problem and many realistic problems can be modeled as a knapsack problem. In this article, four algorithms, more specifically, branch and bound, approximation, hill climbing, and simulated annealing, are implemented and their performance is evaluated on 31 data instances. It is shown that branch and bound have the lowest relative error but a relatively high time complexity compared to that of dynamic programming, and has difficulty solving strongly correlated cases. Hill climbing and simulated annealing are capable of finding a relatively good solution with fast speed, especially in some large-scale examples. The performance of hill climbing relies on a good initialization of the state and it is easy to get stuck into a local optimal. To make up for this limit, simulated annealing adds a stochastic exploration mechanism and it helps to find a better and stable solution given enough time.

KEYWORDS

Knapsack problem, Branch and bound, Dynamic Programming, Simulated annealing, Hill climbing

1 INTRODUCTION

1.1 Literature Review

Knapsack problem (KSP) is a classical combinatorial optimization problem. It usually arises in resource allocation where one needs to choose from a set of resources or tasks under a fixed budget or time constraint, respectively. Over recent decades, it has been the subject of rigorous investigation. A wealth of both precise and heuristic strategies for addressing this problem are documented in the academic discourse, with [8, 15] provided a detailed introduction of different algorithms to solve the knapsack problem.

An interesting characteristic of the 0-1 Knapsack problem is the variation in computational challenge it presents across different instances: while some can be swiftly resolved despite a large number of items, others remain unsolvable with merely a few hundred items [11]. Martello and Toth's book [9] investigates various instances characterized by differing degrees of correlation. They found that instances with no or weak correlation can often be solved more easily. However, instances that exhibit strong correlation, whether they are nearly or inversely correlated, present significant challenges, especially as the size of the problem increases.

There are many algorithms to solve the knapsack problem, which can be divided into three categories:

(1) exact methods such as branch and bound [2, 12, 1, 10] and dynamic programming [14];
(2) approximation method, which includes greedy, Ibarra–Kim algorithm [3] and Fully Polynomial Time Approximation Scheme (FPTAS) [5];
(3) heuristic methods, which include hill climbing, simulated annealing, search and so on.

With the popularity of machine learning, some works introduced neural networks to solve combinatorial problems like KSP. For example, in [13], the authors introduced a heuristic solver based on neural networks for the knapsack problem. This method can only solve small-scale problems and there is a high time complexity for training and testing. The performance can not win over the aforementioned classical methods.

1.2 Our Work

In this article, we compared the following four approaches to solve the Knapsack problem:

(1) Branch and bound
(2) Fully Polynomial Time Approximation
(3) Local search-Hill climbing
(4) Local search-Simulated annealing

Exact methods can be used for some small-scale problems. For large-scale problems, exact methods have a high time complexity, and thus we appeal to some approximation method and heuristic methods in order to find a sub-optimal solution as close to the optimal one as possible.

This article is formulated as follows: in Section 2, we formulate the knapsack problem using mathematical language; In Section 3, we present the details of each algorithm on the KSP. Section 4 shows the empirical evaluation of four proposed algorithms and three groups of plots are provided for local search algorithms to show their effectiveness. Section 5 compares and summarizes the advantages and disadvantages of each algorithm. Finally, Section 6 concludes this article.

2 PROBLEM DEFINITION

The knapsack problem can be formulated as follows: given n items where item i has a non-negative weight w_i and a non-negative value v_i, select a subset of items S such that the total value $\sum_{i \in S} v_i$ is maximized and the total weight $\sum_{i \in S} w_i$ should not exceed a weight limit W. Mathematically, we can formulate it as the following

combinatorial optimization problem with an inequality constraint:

$$\max \sum_{i=1}^{n} x_i v_i \tag{1}$$
$$s.t., \sum_{i=1}^{n} x_i w_i \leq W$$
$$x_i \in \{0, 1\},\ i = 1, 2 \cdots n$$

3 ALGORITHMS

3.1 Branch and bound

The comprehensive Branch and Bound algorithm employed for the 0/1 Knapsack problem is outlined as Function *KnapsackBnB* in Algorithm 1. Consistent with the conventions discussed in our lectures, we preserve the standard definitions of upper and lower bounds.

To enhance pruning efficiency, we initialize the upper bound using results from an Approximation algorithm. This initialization reflects the global maximum of the total value achievable, setting a benchmark for potential solutions.

The method to calculate the lower bound for each node configuration, denoted as *GetBound*, follows the approach described by Peter [6]. This function computes the lower bound by incorporating the strategy of the fractional knapsack problem, which permits the partial inclusion of items. This allows each node to estimate the maximum potential value it can achieve if it were expanded completely. The goal of the *GetBound* function is achieving a tighter bound and thus enabling more effective pruning of branches.

Branching within the algorithm, as facilitated by the *Branch* function, employs the "best first criterion." This criterion prioritizes nodes based on their lower bounds, with the priority queue in *KnapsackBnB* ordering nodes from the highest to the lowest potential value. Nodes that are either leaf nodes or reach the knapsack's capacity are evaluated to determine if they surpass the current global maximum value. As the algorithm expands nodes, any child node with a lower bound that falls below the current upper bound is pruned. This strategy significantly enhances the efficiency of the algorithm by focusing computational resources on the most promising branches and discarding less promising ones early in the process.

Since the branch and bound algorithm evaluates the inclusion of each individual item, its time and space complexities are both $\mathbf{O}(2^n)$, where n is the number of items. The accuracy of this algorithm is a key strength, as it exhaustively explores all possibilities. Nevertheless, this thoroughness is a double-edged sword; the running time not only shows expected exponential growth with respect to the input size but is further exacerbated when the input instances are strongly correlated [9], presenting a considerable challenge, particularly for large inputs.

3.2 Approximation: FPTA

To address these challenges, this implementation uses a Fully Polynomial Time Approximation Scheme (FPTAS), which provides a way to obtain solutions that are close to optimal within a user-defined error margin ϵ. This approach is particularly valuable in

Algorithm 1: Branch and Bound algorithm for the 0/1 Knapsack problem

```
1  Function KnapsackBnB(capacity, weights, values)
       input  : capacity of the knapsack, weights and values of items
       output : max value obtainable and the items taken
2      Initialize max_val and selected_items with Approximation;
3      Sort items in descending order of v/w;
4      Initialize a min-heap priority queue pq ordered by the negative value of the bound;
5      sentinel ← new node with −1 index and 0 valueSum;
6      Add sentinel to pq with a default priority;
7      while pq is not empty do
8          cur ← pq.pop();
9          Branch(cur, n, weights, values, pq);
10     return Solution constructed from the global best node;
11 Function Branch(node, n, weights, values, pq)
12     if node.last_add_item_index ≥ len(items) - 1 then
13         if node.weightSum ≤ capacity and node.valueSum > max_val then
14             max_val ← node.valueSum;
15             selected_items ← node.items_included;
16         continue;
17     if node.weightSum == capacity and node.valueSum > max_val then
18         max_val ← node.valueSum;
19         selected_items ← current.items_included;
20     next_id ← idx + 1;
       // Create and evaluate child nodes for including and excluding the next item
21     for option in {include, exclude} do
22         child ← Node(node, next_id, option, weights, values);
23         child.lb ← GetBound(n, child, weights, values, capacity);
24         if child.lb > max_val then
25             Add child to pq with priority as −child.lb;
26 Function GetBound(n, child, weights, values, capacity)
27     if child.weightSum > capacity then
28         return 0;
29     lb ← node.valueSum;
30     j ← node.last_add_item_index + 1;
31     total_weight ← node.weightSum;
32     while j < n and total_weight + weights[j] ≤ capacity do
33         total_weight ← total_weight + weight[j];
34         lb ← lb + values[j];
35         j ← j + 1;
36     if j < n and total_weight < capacity then
37         capacityLeft ← capacity - total_weight;
38         lb ← lb + (capacityLeft × values[j] / weights[j]);
39     return lb;
```

scenarios where an exact solution is less critical than achieving a near-optimal solution quickly.

The FPTAS for the knapsack problem works by scaling down the values of the items based on the maximum value and the approximation factor ϵ, which reduces the problem's complexity while controlling the quality of the solution:

- **Scaling Factor**: The values are scaled by a factor determined by ϵ and the maximum item value, reducing the range of possible total values and simplifying the problem.
- **Dynamic Programming Solution**: A dynamic programming algorithm is applied to the scaled problem, which is computationally less intensive due to the reduced value range.
- **Result Scaling**: After solving the scaled problem, the solution is scaled back to approximate the solution for the original problem values.

It is a $1+\epsilon$ approximation for the knapsack problem with the proof in [17]. Both the space and time complexities [17] are $\mathbf{O}(nW)$ (in order to perform the traceback, the space cannot be optimized to $\mathbf{O}(W)$).

A description of our logic using pseudo-code, solving the given knapsack problem can be seen in Algorithm 2.

Algorithm 2: Fully Polynomial Time Approximation Scheme (FPTAS) for the 0/1 Knapsack Problem

Function FPTASKnapsack(*(items, maxWeight, epsilon)*)**:**
  $maxValue \leftarrow$ max(items.value);
  $n \leftarrow$ length(items);
  $K \leftarrow maxValue \times \epsilon / n$;
  $scaledItems \leftarrow$ list of{value $\leftarrow$ $\lfloor$item.value$/K\rfloor$, weight $\leftarrow$ item.weight};
  // Initialize dynamic programming table
  $dp \leftarrow$ array$[0 \ldots n][0 \ldots maxWeight]$ initialized to 0;
  for $i \leftarrow 1$ **to** n **do**
    for $weight \leftarrow 0$ **to** $maxWeight$ **do**
      if $item.weight \leq weight$ **then**
        $dp[\mathrm{i}][weight] \leftarrow \max(dp[\mathrm{i}-1][weight], dp[\mathrm{i}-1][weight - item.weight] + item.value)$;
      else
        $dp[\mathrm{i}][weight] \leftarrow dp[\mathrm{i}-1][weight]$;
  // Trace back to find the items included in the knapsack
  $approximateTotalValue \leftarrow \lfloor dp[n][maxWeight] \times K \rfloor$;
  $selectedItems \leftarrow$ empty list;
  $w \leftarrow maxWeight$;
  for *item from scaledItems in reverse* **do**
    if $dp[item.index][w] \neq dp[item.index-1][w]$ **then**
      $selectedItems$.add$(item)$;
      $w \leftarrow w - item.weight$;
  return $(approximateTotalValue, selectedItems)$;

The code outlined in Algorithm 2 provides functionality to parse problem data from files, implement the FPTAS algorithm, and validate the computed solutions against official benchmarks. This design emphasizes efficiency and flexibility, making it ideal for educational, research, and practical scenarios that require quick approximation solutions. Its main advantage lies in its rapid execution and high accuracy. However, a notable limitation is that the maximum value, derived from the scaled item values, yields indices that lack meaningful interpretation. This issue may impact the usability of the results in certain applications where item indices are significant.

3.3 Local search: hill climbing

Hill climbing (HC) is an optimization technique which belongs to the family of local search. This algorithm finds a satisfactory solution to the Knapsack problem without having to look at all possible combinations, which is especially useful when dealing with a large number of items.

Our HC algorithm starts with a randomly generated knapsack configuration, created by a binomial distribution. Given a random configuration of the initial state, we check whether the initial state already exceeds the total weight expectation. If so, we regenerate the initial state until it meets the weight requirement. The time consumed on regenerating initial states is also counted as a part of the algorithm's running time. We set up an evaluation step to calculate the total weight and total value of the items in the current knapsack.

The critical step is to generate neighbors of the current state by making small changes to the current state (e.g., adding or removing a single item). During each iteration, the HC algorithm reviews all generated neighboring states and selects the one that offers the highest increase in total value without surpassing the predefined weight limit. This process of selecting and moving to the best neighboring state continues iteratively. The algorithm halts its execution when it encounters a state where no single neighbor provides an improvement in value within the weight constraints. The most optimal knapsack configuration found through this method is then deemed the final output.

The pseudocode for the hill climbing algorithm shown in Algorithm 3, clearly describes the sequence of operations performed from initialization to termination of the algorithm.

Compared with the default hill climbing algorithm, we made two major changes. Instead of starting with an empty knapsack, we begin with a randomly generated knapsack configuration, created using a binomial distribution. This can decently reduce the relative error of the algorithm in practice. Another attempt we made was to generate many neighbors (e.g. 100) for a current state and choose the one that offers the highest increase in total value without surpassing a predefined weight limit. This often increases the accuracy of the algorithm but can lead to a much longer running period. After the trade-off consideration, we decide to adopt the default choice with only one neighbor as our final solution for this project.

One concern related to the initial non-empty knapsack arises when the initial knapsack has the possibility of exceeding the weight limit. We add a step of checking whether the initial state already exceeds the total weight expectation. If so, we regenerate

initial state until it exhibits a weight no larger than the weight requirement. After tuning, we decide to apply a binomial distribution where the probability of selecting 1 is set at 0.1 for smaller datasets and 0.01 for larger ones, which is a reasonable choice for this specific context. If the initial state is overloaded with too many 1s, the algorithm can be stuck for an extremely long time in attempting to find an initial state that does not surpass the expected total weight. In the worst scenario, it could become permanently stuck if no suitable initial state meets the weight requirements. After many rounds of trials, we set the probability of selecting 1 at 0.1 for smaller datasets and 0.01 for larger ones in the binomial generation process to avoid the situation. In practice, the alternative way is to run the algorithm for a large number of times (e.g., 100) without considering the weight limit of the initial state, but store and compare only those results that fall under the weight limit. In this way, we can avoid spending too much time on the attempts to find an appropriate initial state. The robustness of the generated solution is not significantly diminished. This alternative method works especially well for large datasets where the generation of initial state could be stuck for quite a while.

3.4 Local search: simulated annealing

Simulated annealing (SA) is an effective local search algorithm. Different from hill climbing, it accepts a bad choice with some probability which enables the algorithm to get out of a local optimal. If the selected move improves the solution, it is accepted, otherwise, the algorithm makes a move with some probability less than 1.

Denote X as a 0-1 vector with length n where each item x_i denotes whether the i-th term is chosen or not. Denote $cost(X) = \sum_{i=1}^{n} x_i v_i$ which satisfies the constraint $\sum_{i=1}^{n} x_i w_i \leq W$. The probability of accepting a new solution X' over the old one X is

$$P(X', X) = \begin{cases} 1, \ cost(X') < cost(X) \\ \exp\left(\frac{cost(X') - cost(X)}{T}\right), \ \text{otherwise} \end{cases} \tag{2}$$

where T is a pre-defined temperature.

The pseudo-code of the simulated annealing algorithm is shown in Algorithm 4. The function *runSA* is the main function of the algorithm which defines a pre-defined temperature, decay rate of the temperature, iteration numbers of each temperature, and initiation of the initial state X. To make sure the program generates a legal initial state, we use the binomial distribution and re-generate another state if the recent one is illegal. The function *SA* is the implementation of the simulated annealing algorithm where α denotes the decaying rate of the temperature. The function *cost* is used to evaluate the total value obtained with the current X. The function *generateNeighbor* generates a valid new state X' by randomly flipping one item in the old X. The algorithm stops when the average change in the objective function is less than a tolerance value or when the temperature reaches the minimum temperature (>0).

4 EMPIRICAL EVALUATION

4.1 Setup and Test Cases

Table 1 is the description of our platform (CPU, RAM, and used programming language). Despite different setups, the results are comparable.

Algorithm 3: Hill Climbing Algorithm for the 0/1 Knapsack Problem

```
Function runHC (n, W, values, weights, iterations, numTrials):
    Initialize best_result to a very low value;
    Initialize best_solution to None;
    Initialize results to an empty list;
    for i ← 1 to numTrials do
        X0 ← generates a random initial solution with binomial distribution;
        while total weight of X_0 is greater than W do
            X_0 ← generate new random initial solution;
        bestX, bestValueWeight, trace_data ← runHC(X0, values, weights, W, iterations);
        Append bestValueWeight[0] to results;
        if bestValueWeight[0] > best_result then
            best_result ← bestValueWeight[0];
            best_solution ← bestX;
    return best_solution, best_result, results;
Function HC (X0, values, weights, W, maxiter):
    X ← X0;
    bestX ← X0;
    bestValue, bestWeight ← cost (X, values, weights);
    for i ← 1 to maxiter do
        newX ← generateManyNeighbors (X);
        for neighbor in newX do
            newValue, newWeight ← cost (neighbor, values, weights);
            if newWeight ≤ W and newValue > bestValue then
                X ← neighbor;
                bestX ← neighbor;
                bestValue ← newValue;
    return bestX, (bestValue, bestWeight);
Function cost (X, values, weights):
    totalValue = Σ_{i=1}^{n} X_i · values_i;
    totalWeight = Σ_{i=1}^{n} X_i · weights_i;
    return totalValue, totalWeight
Function generateNeighbors (X):
    for chosenInd from 1 to n do
        X' = X;
        X'[chosenInd] = 1 − X[chosenInd];
        yield X'
Function generateManyNeighbors (X):
    neighbourhood ← [];
    m ← desired number of neighbors;
    for i ← 0 to m do
        temp ← copy of X;
        if temp[i][i] = 1 then
            temp[i][i] ← 0;
        else
            temp[i][i] ← 1;
        Append temp to neighbourhood;
    return neighbourhood;
```

Algorithm 4: Simulated annealing for KSP

```
Function runSA (InputFile, SolFile, OutputFile):
    X_0 ← random.binomial();
    sol = SA (iterPertemparature, X_0, values, weights, W, T_0, α);
    return sol
Function SA (iterPertemparature, X_0, values, weights, W, T0, α):
    T ← T_0, X_best ← X_0, X ← X_0, i ← 0 ;
    currtotalValues = cost (X, values, weights).totalvalues;
    while T>minTemperature do
        i = 0;
        while i < iterPertemperature do
            while True do
                X′ = generateNeighbor(X);
                if cost (X′).totalweights<W then
                    break
            if cost (X′).totalvalues>currtotalValues then
                X = X′;
                currtotalValues ← cost(X′).totalvalues
            else
                r = random(0, 1);
                if r < exp((cost(X′) − cost(X)))/T then
                    X ← X′, ;
                    currtotalValues = cost (X′).totalvalues
            if currtotalValues > maxTotalValues then
                X_best ← X ;
                maxTotalValues ← currtotalValues
            i ← i + 1
        T ← αT
    return X_best, maxTotalValues
Function cost (X, values, weights):
    totalValue ← Σ_{i=1}^{n} X_i · values_i ;
    totalWeight ← Σ_{i=1}^{n} X_i · weights_i ;
    return totalValue, totalWeight
Function generateNeighbor (X):
    random choose an index from 1 to n, denoted as chosenInd ;
    X′ ← X ;
    X′[chosenInd] ← 1 − X[chosenInd];
    return newX
```

Our evaluation framework comprises two distinct sets of test cases: small and large scale. The small-scale set encompasses 10 test cases, with input sizes varying from 4 to 23, featuring instances that exhibit varying degrees of correlation. The large-scale set consists of 21 test cases, with input sizes expanding from 100 to 10,000. These are further categorized into three subsets: `large_1` to `large_7`, `large_8` to `large_14`, and `large_15` to `large_21`. Within these subsets, the input sizes ascend in a structured sequence: 100, 200, 500, 1,000, 2,000, 5,000, and 10,000.

We intend to execute all four algorithms across these varied test cases to conduct a comparative analysis of the outcomes.

Name	CPU	RAM	Language
Yifan Liu	MacBook M2	16GB	Python
Qidi Gao	MacBook M2	16GB	Python
Andi Xia	Intel Xeon Gold 6226 2.70GHz	192GB	Python
Mi Zhou	AMD Ryzen 7 5800H 3.20 GHz	16GB	Python

Table 1: Configuration of our personal computers.

4.2 Empirical Resullts

Table 2 and 3 summarize the comparison of different algorithms based on the time complexity, the total value obtained, and the relative error. The relative error (RelErr) is defined as

$$RelErr = \frac{OPT - Alg}{OPT}, \tag{3}$$

where OPT represents and optimal solution of the problem and Alg represents the solution obtained by proposed algorithms.

When using local search algorithms (i.e., hill climbing and simulated annealing), we use three measures to evaluate the performance of these algorithms:

(1) QRTD: qualified runtime for various solution qualities. QRTD focuses on the running time distribution of an algorithm conditional on its success in achieving a certain level of solution quality q' to a given problem π'. In this problem, it is defined as $qrtd_{q'}(t) := rtd(t, q') = P_s(RT \leq t, RelErr \leq q')$ where RT denotes the time instance when solution improved, t is the cutoff time, $RelErr$ is the relative error, q' is a set bound.
(2) SQD: solution quality distribution for various run-times. SQD focuses on the probability distribution of the quality of solutions achieved by an algorithm within a given time frame. It is defined as $qrtd_{t'}(q) := rtd(t', q) = P_s(RT \leq t', RelErr \leq q)$.
(3) Box plots for running times. The box plot shows five important information in the data: upper whisker, upper quartile, median, lower quartile, and lower whisker. It can be used as a graph summarization of the distribution of the running time.

4.2.1 Branch and Bound. For test cases with small input sizes, as depicted in Table 2, the branch-and-bound algorithm resolves most problems swiftly, typically within 0.01 seconds, and achieves a relative error of zero. The running times do not display a distinct pattern due to the small input size. However, in test cases such as `small_8`, the efficacy of the algorithm's "best first criteria" is compromised. We suspect this is because, with the strong correlation of instances, the criteria cannot effectively distinguish between items. Concurrently, the lower bounds of each node are often very close to the globally best value found, leading to minimal pruning and necessitating a near-complete search.

For test cases with larger input sizes, as delineated in Table 3, the branch and bound algorithm demonstrates a rate of running times that not only increases but also exhibits exponential growth

for more complex test cases. The running time plot in Figure 1 illustrates these relationships:

- Groups `large_1` to `large_7` and `large_8` to `large_14` exhibit similar running time patterns, which causes their respective curves to overlap in the plot. Their running times increase with a small increasing rate, and this growth rate itself accelerates slightly as input sizes grow, benefiting from the initial upper bounds established by the approximation algorithm.
- For the group spanning `large_15` to `large_21`, a time limit of 3,600 seconds was imposed for cases `large_19` to `large_21`. Due to the excessive core memory requirements, we did not extend the running time, leaving their exact resolution times undetermined; it remains unknown whether these cases are solvable within a practical time frame. Nevertheless, the observed running time pattern strongly suggests an exponential increase. These observations lead us to conclude that this is similar to the case of `small_8`, where strong item correlations cause the branch-and-bound algorithm to falter in effective branching and pruning.

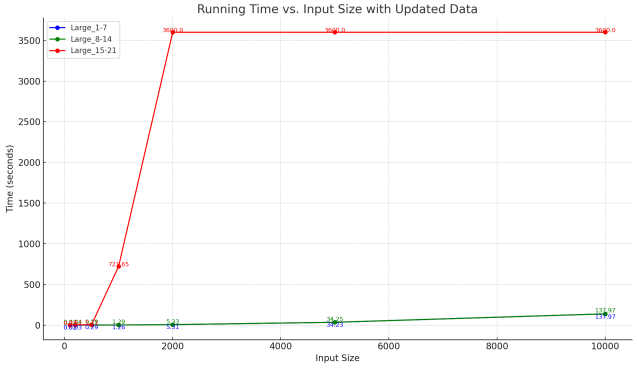


Figure 1: Running time of three groups of large test cases

4.2.2 Approximation. This section presents an empirical evaluation of the approximation algorithm, tested on both small-scale and large-scale datasets to assess its performance. The evaluation primarily focuses on the time efficiency and approximation accuracy of the algorithm to determine its applicability to knapsack problems of varying sizes. ϵ is set as 0.1 in our analysis.

On the small-scale datasets, the algorithm demonstrated high approximation accuracy, with approximation ratios mostly close to 1.0, indicating that the algorithm can generate solutions nearly optimal compared to the official best solutions. For each dataset, the processing time remained within reasonable limits, confirming the efficiency of the algorithm in handling smaller problems. Here is a brief summary of specific results:

- For datasets `small_1` to `small_10`, the approximation ratios ranged from 0.9898 to 1.0, showing consistency in maintaining solution quality.
- Processing times varied from 0.0002 seconds to 0.0288 seconds, illustrating that the algorithm is capable of quickly producing results even on smaller datasets.
- Notably, dataset `small_5` showcased the robustness of the algorithm in handling data with non-integer values, despite a minor approximation error.

On large-scale datasets, the algorithm also showed high accuracy and acceptable time efficiency, although computation time significantly increased with problem size. Here are detailed evaluations for datasets `large_1` to `large_21`:

- Approximation ratios were close to 1.0 in most cases, with the lowest being 0.9958, indicating that the algorithm can generate high-quality solutions for larger-scale problems as well.
- Processing time increased with the size of the data, ranging from 0.0052 seconds to 50.6398 seconds, reflecting the computational burden on more complex problems.
- Particularly for datasets like `large_6` and `large_7`, despite longer processing times, the algorithm maintained high approximation ratios, demonstrating good scalability and robustness.

4.2.3 Local search. While the Branch & Bound algorithm guarantees the optimal solution, it can take a very long time to finish the implementation, especially for large datasets. Instead, we explore two local search methods: hill climbing and simulated annealing.

HC algorithm is very fast in finding a solution. The larger the dataset is, the more obvious this advantage can be. As a trade-off, the generated solution might not be highly satisfying, with the relative error ranging from 0.15 to 0.89. After many adjustments and tests, we find the performance of the hill climbing algorithm in this specific task likely depends on the choice of the initial state. HC algorithm iteratively makes adjustments to the default solution, with the goal of finding a better solution by incrementally changing a single element of the solution. If we propose the initial state as an empty set, for the problem of knapsack, it can easily get stuck at a local maximum. These are points where all immediate neighbors have worse values, but they are not decent solutions overall as the global optimum. When the initial state is not empty but a random set generated by binomial distribution, the algorithm can often achieve a relatively decent performance with the initial state and reach better performance after several rounds of improvement. This phenomenon might depend on the nature of the knapsack problem.

The actual running time for our HC algorithm includes the time for finding qualified initial states that do not exceed the weight limit, and the time for finding better neighbors to replace and improve the solution. While the number of hills is little known and barely controllable due to the random set-up of these datasets, we find large scale problem takes more time than that of small scale problems in general. One of our attempts also demonstrate finding many neighbors are more time-consuming than finding only one neighbor.

Given the limitation of hill climbing in accuracy improvement, we then move to the simulated annealing algorithm that allows for occasional worse moves to escape local optima.

For the simulated annealing algorithm, the time complexity depends on several parameters:

(1) The scale of the problem: large scale problem takes more time than that of small scale problems

(2) The temperature T and its decay rate.
(3) The iteration number used for each temperature.

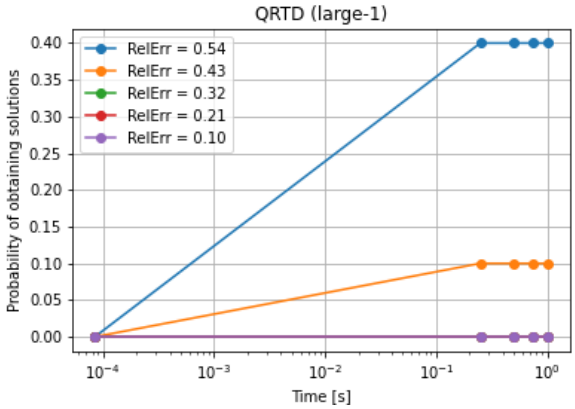
Here we choose the parameters based on the compromise of time complexity and accuracy. Each experiment is run 20 times with different random seeds and the average time, average total values, and average relative error are calculated. The minimum temperature is set up as $T_{\min} = 5$. For the small-scale problems, we fix $T = 1000, \alpha = 0.95$, $iterPerTemparature = 50$, and set X_0 as a zero vector which means no item is chosen. For the large-scale problem, the parameters are $T = 1000$, $iterPerTemparature = 200$, $\alpha = 0.95$ to compromise between the time and the relative error. As we can see from Table 2, the SA algorithm almost finds the global optimal solution for all the instances within 0.15 seconds. For the large-scale instances, the time increases with the scale of the problem increases. `large_7`, `large_14`, `large_21` have 10,000 items and take around 300 seconds on average with high relative errors around $0.2 \sim 0.26$. The time and relative error tend to increase when the scale of the problem increases, which is under expectation since SA is a search-based algorithm. Different from the HC algorithm, the SA algorithm does not rely on a good initialization of the state. The relative error decreases if more iterations are given.

To further evaluate the local search algorithms, we plot the QRTD, SQD, and running time boxplot on instances `large_1` and `large_3` based on 20 experiment runs with different initialization seeds. The large-scale 1 has 100 items and the large-scale 3 has 500 items. Thus, we set the cutoff time as $t = 10$ seconds for `large_1` and $cutoff = 20$ for `large_3`.

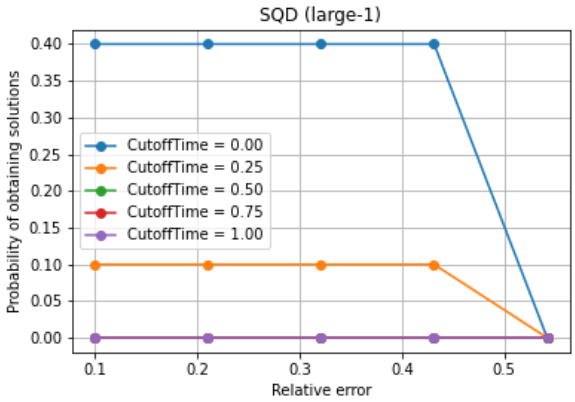
Each experiment is run 20 times to obtain 20 trace files logging down the time and the quality of the solution when the algorithm finds a better solution. Fig. 2 and Fig. 3 are the three evaluation plots (QRTD, SQD, running time) for the HC algorithm on two instances: `large_1` and `large_3` respectively. Fig. 4 and Fig. 5 are the three evaluation plots (QRTD, SQD, running time) for the SA algorithm on two instances: `large_1` and `large_3` respectively. As we can see from Fig. 2(c), Fig. 3(c), Fig. 4(c), and Fig. 5 (c), the running time of local search algorithm has a high variance. The interquartile range (i.e., IQR [1]) of the HC algorithm is larger than that of the SA algorithm. This phenomenon corroborates the strong dependence of the HC algorithm on the initial states.

The QRTD plot for the SA algorithm 4(a), Fig. 5 (a) show that the probability of obtaining is getting lower if the relative error is set smaller. For example, in the data instance `large_1`, when the relative error is set as 0.1 (purple curve), the probability of obtaining the solution is almost 1. However, when the relative error is set as 0.03 (orange curve), it needs more cutoff time to make the probability get to 1. The QRTD plots for the HC algorithm show a similar trend. The SQD plots for the SA algorithm Fig. 4(b), Fig. 5 (b) show that fixing the cutoff time, lower relative error implies a lower probability of obtaining the solution. This phenomenon, however, is not shown in the SQD plots for the HC algorithm. For example, in Fig. 2(b) and Fig. 3(b), we observe a low probability of obtaining the solution corresponds to a high relative error given a fixed cutting time. This may relate to the fact that the HC algorithm gets stuck into a local optimal starting from a bad initialization state. Thus, giving more running time won't improve the results.

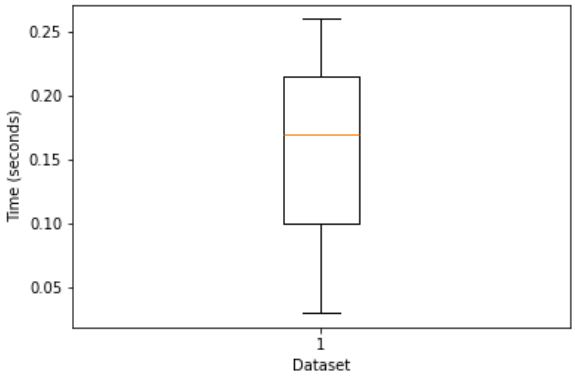
[1]The IQR is defined by the distance between the upper quartile and the lower quartile



(a)



(b)



(c)

Figure 2: Hill climbing algorithm for large-scale 1: (a) QRTDs (b) SQDs (c) Running time

Table 2: Small scale problems: Comparison of different algorithms on the knapsack problem

Dataset	Branch and Bound			FPTA			Hill climbing			Simulated annealing		
	Time (s)	Total value	RelErr	Time (s)	Total value	RelErr	Time (s)	Total value	RelErr	Time (s)	Total value	RelErr
small_1	0.01	295	0	0.0023	292	0.0102	0.041	230.00	0.190	0.12	295	0.00
small_2	0.01	1024	0	0.0043	1019	0.0049	0.047	887.20	0.134	0.12	1023.7	0.00029
small_3	0.00	35	0	0.0002	34	0.0286	0.036	32.80	0.060	0.094	35	0.00
small_4	0.00	23	0	0.0002	22	0.0435	0.036	19.65	0.146	0.10	23	0.00
small_5	0.01	481.0694	0	0.0009	478	0.0064	0.043	399.80	0.169	0.12	481.069	6.65e-08
small_6	0.01	52	0	0.0009	52	0.000	0.040	46.75	0.101	0.11	52	0.00
small_7	0.00	107	0	0.0002	107	0.000	0.039	91.80	0.142	0.11	107	0.00
small_8	127.42	9767	0	0.0288	9746	0.0021	0.049	9655.8	0.011	0.16	9744.5	0.0023
small_9	0.00	130	0	0.0002	128	0.0154	0.037	114.25	0.121	0.094	130	0.00
small_10	0.01	1025	0	0.0021	1020	0.0049	0.047	901.85	0.120	0.12	1023.8	0.0012

Table 3: Large scale problems: Comparison of different algorithms on the knapsack problem

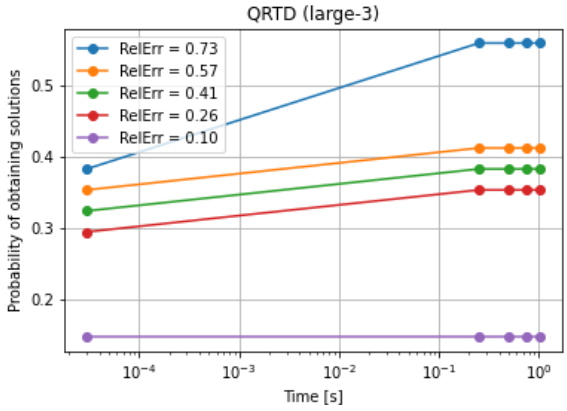
Dataset	Branch and Bound			FPTA			Hill climbing			Simulated annealing		
	Time (s)	Total value	RelErr	Time (s)	Total value	RelErr	Time (s)	Total value	RelErr	Time (s)	Total value	RelErr
large_1	0.02	9147	0	0.011	9141	0.0007	0.153	5530.95	0.395	2.69	6368.85	0.0073
large_2	0.03	11238	0	0.016	11233	0.0004	0.274	5491.00	0.511	6.10	11116.6	0.011
large_3	0.29	28857	0	0.116	28852	0.0002	0.647	24417.30	0.154	12.54	27590.55	0.044
large_4	1.26	54503	0	0.461	54499	0.0001	0.998	46465.61	0.147	24.88	50753.95	0.069
large_5	5.31	110625	0	1.897	110625	0	1.361	19422.10	0.824	49.18	99520.4	0.10
large_6	34.23	276457	0	12.55	276457	0	3.798	43429.63	0.843	156.58	231504.8	0.16
large_7	137.97	563647	0	49.60	563647	0	7.370	61498.80	0.891	301.62	421635.8	0.25
large_8	0.02	1514	0	0.006	1508	0.004	0.125	1183.62	0.218	3.17	1458.8	0.036
large_9	0.04	1634	0	0.010	1631	0.002	0.250	1368.20	0.162	7.34	1536.15	0.059
large_10	0.29	4566	0	0.102	4562	0.001	0.532	2889.38	0.367	14.14	4064.6	0.11
large_11	1.29	9052	0	0.447	9048	0.0004	0.989	5881.67	0.350	27.44	7869.6	0.13
large_12	5.33	18051	0	1.889	18047	0.0002	1.913	12816.45	0.290	54.74	15249.15	0.16
large_13	34.25	44356	0	12.32	44352	0.0001	4.604	28164.32	0.365	143.33	35970.15	0.19
large_14	137.97	90204	0	49.67	90200	0.00004	9.184	60110.22	0.334	347.15	70143.05	0.22
large_15	0.03	2397	0	0.005	2387	0.004	0.130	1795.85	0.251	2.75	2313.45	0.035
large_16	2.52	2697	0	0.011	2692	0.002	0.336	1994.45	0.260	6.38	2567.45	0.048
large_17	5.11	7117	0	0.100	7112	0.001	0.767	4316.88	0.393	13.09	6352.45	0.11
large_18	721.65	14390	0	0.449	14385	0.0003	1.826	7390.50	0.486	24.63	12536.15	0.13
large_19	3600	28473	0.015	1.868	28914	0.0155	2.386	13526.38	0.532	49.88	24260.9	0.16
large_20	3600	71968	0.007	12.29	72500	0.0073	5.123	32605.83	0.550	161.26	57148.85	0.21
large_21	3600	146382	0.004	50.64	146914	0.0036	9.233	61191.72	0.584	312.34	108083.3	0.26

5 DISCUSSION

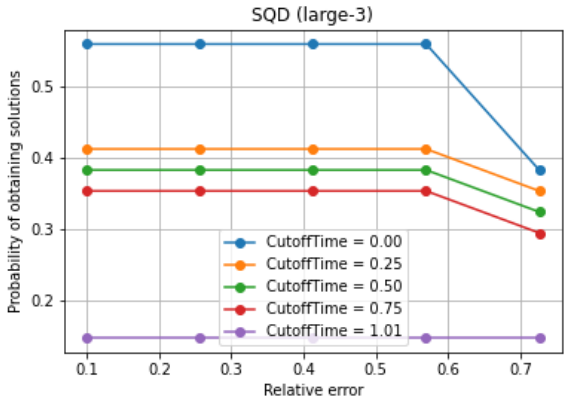
Our results demonstrate the advantages and disadvantages of these algorithms in the context of knapsack problem, as shown in Table 2 and Table 3.

The branch-and-bound algorithm excels in accuracy, consistently delivering deterministic solutions and outperforming three other algorithms in precision. However, its efficiency varies with the input size and the degree of item correlation within instances, with performance notably diminishing when items are highly correlated. Among the four algorithms evaluated, branch-and-bound is uniquely sensitive to instance correlation. Introducing a robust approximation algorithm to establish an initial upper bound could significantly expedite processing times. Additionally, developing advanced strategies for calculating lower bounds that effectively identify promising nodes, particularly in cases of strong correlations, is crucial. Furthermore, the application of parallel computing on GPUs, as explored in studies such as [16, 7], holds the potential to markedly enhance runtime efficiency.

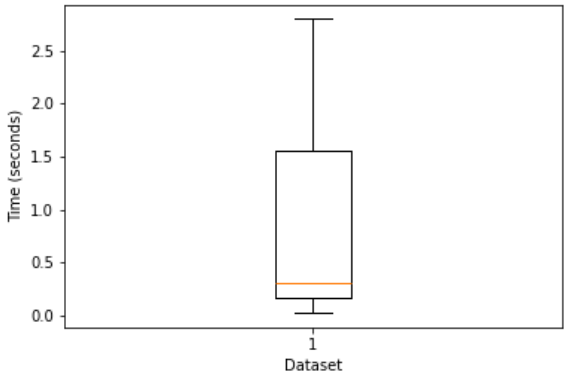
The empirical results of Approximation indicate that the algorithm is capable of efficiently solving knapsack problems of both small and large scales while maintaining solution quality. This performance makes the algorithm suitable for applications requiring quick solutions to complex discrete optimization problems, such as resource allocation and optimal decision support. Further optimization of the algorithm's structure and parameters could potentially enhance its performance on even larger datasets.



(a)

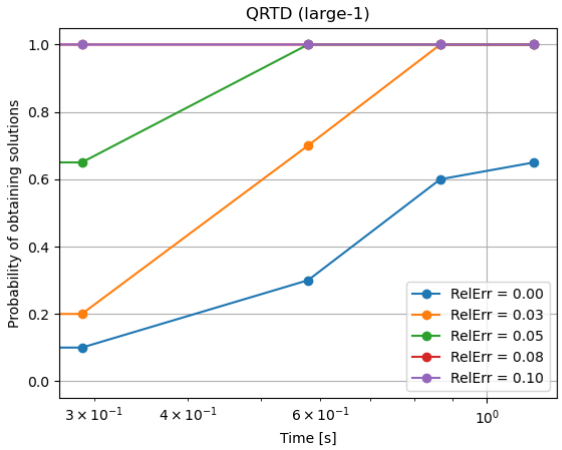


(b)

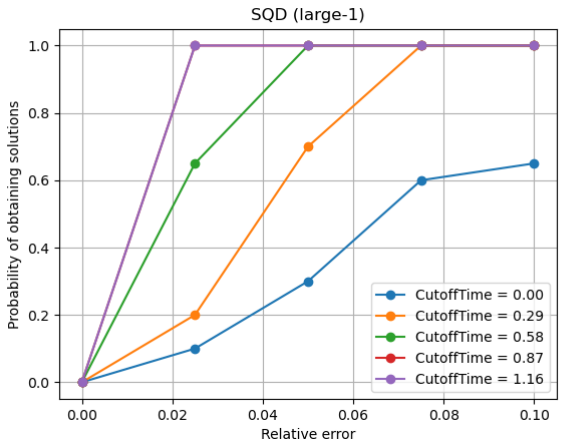


(c)

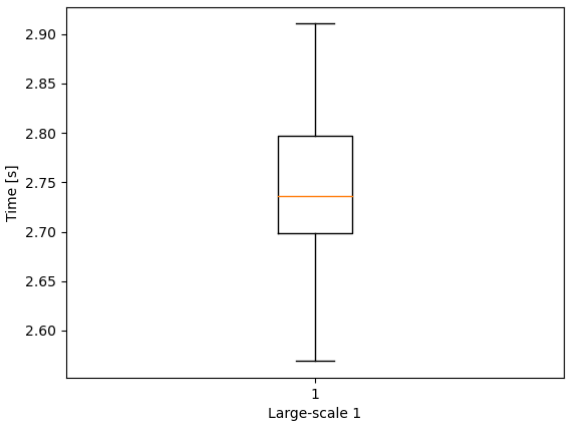
Figure 3: Hill climbing algorithm for large-scale 3: (a) QRTDs (b) SQDs (c) Running time



(a)

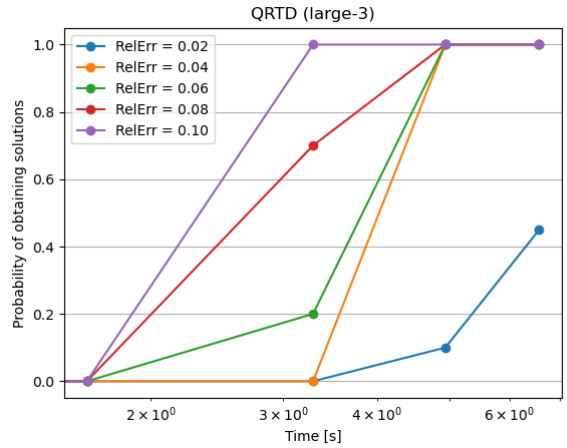


(b)

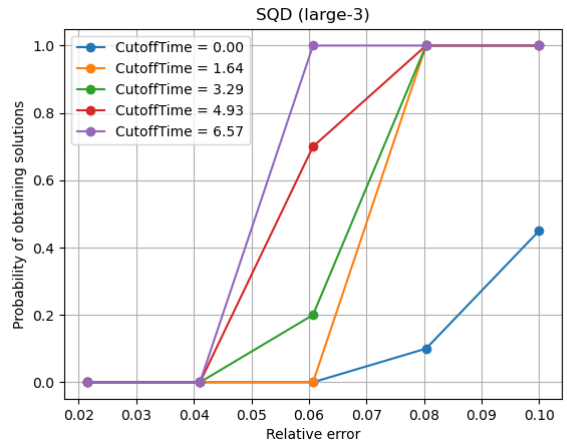


(c)

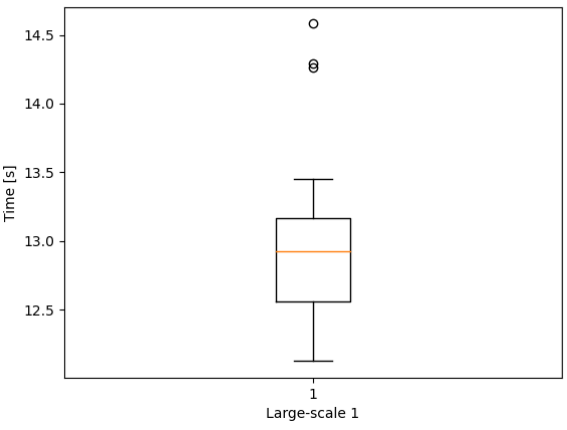
Figure 4: Simmulated annealing algorithm for large-scale 1: (a) QRTDs (b) SQDs (c) Running time boxplot



(a)



(b)



(c)

Figure 5: Simmulated annealing algorithm for large-scale 3: (a) QRTDs (b) SQDs (c) Running time boxplot

In terms of local search algorithms, we implement hill climbing and simulated annealing algorithms. Hill climbing is good at finding a solution very quickly, but can easily get trapped into a local optimal that is far away from the global optimal. Our customized version of hill climbing generate the initial state with a binomial distribution, significantly increasing the accuracy of the algorithm [4]. Simulated annealing, obtains consistently better performance especially for the large scale dataset. The good performance is achieved at the expense of substantial running time.

6 CONCLUSION

This study has explored the effectiveness of various algorithms in solving the Knapsack problem, focusing on both small-scale and large-scale datasets. Through rigorous empirical evaluation, we have assessed four main algorithms: Branch and Bound, Fully Polynomial Time Approximation Scheme (FPTAS), Hill Climbing, and Simulated Annealing.

Our results demonstrate that the Branch and Bound method, while precise, faces limitations in scalability and time efficiency as the problem size and correlations in instances increase. The FPTAS provided a good balance between performance and accuracy, proving to be a reliable approach for larger datasets. On the other hand, heuristic methods like Hill Climbing and Simulated Annealing introduced flexibility and performed reasonably well, especially when traditional methods were computationally prohibitive.

Each algorithm has shown distinct strengths and weaknesses, making them suitable for specific types of knapsack problems. For instance, Hill Climbing and Simulated Annealing are preferable for problems where approximate solutions are acceptable and rapid execution is required. In contrast, Branch and Bound is ideal for scenarios where accuracy cannot be compromised and the problem is of small or medium size.

Future work could explore hybrid approaches that combine the robust accuracy of exact methods with the flexibility and speed of heuristic techniques. Such hybrid methods could potentially offer a more holistic solution to the knapsack problem, accommodating both the need for speed and precision. Additionally, integrating machine learning models to predict algorithm performance based on problem characteristics could further optimize the selection process for an appropriate solving strategy.

In conclusion, this comprehensive study not only highlights the capabilities and limitations of each algorithm but also sets the stage for future innovations in solving complex optimization problems like the knapsack problem.

REFERENCES

[1] Didier Fayard and Gérard Plateau. 1982. Algorithmus 47. ein algorithmus für die lösung des 0–1 knapsack problems. *Computing*, 28, 269–287.

[2] Ellis Horowitz and Sartaj Sahni. 1974. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21, 2, 277–292.

[3] Oscar H Ibarra and Chul E Kim. 1975. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)*, 22, 4, 463–468.

[4] David Iclanzan and Dan Dumitrescu. 2007. Overcoming hierarchical difficulty by hill-climbing the building block structure. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 1256–1263.

[5] Hans Kellerer and Ulrich Pferschy. 1999. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3, 59–71.

[6] Peter J Kolesar. 1967. A branch and bound algorithm for the knapsack problem. *Management science*, 13, 9, 723–735.

[7] Mohamed Esseghir Lalami and Didier El-Baz. 2012. Gpu implementation of the branch and bound method for knapsack problems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1769–1777.

[8] Silvano Martello and Paolo Toth. 1987. Algorithms for knapsack problems. In *Surveys in Combinatorial Optimization*. North-Holland Mathematics Studies. Vol. 132. Silvano Martello, Gilbert Laporte, Michel Minoux, and Celso Ribeiro, (Eds.) North-Holland, 213–257. DOI: https://doi.org/10.1016/S0304-0208(08)73237-7.

[9] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.

[10] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.

[11] Silvano Martello and Paolo Toth. 1997. Upper bounds and algorithms for hard 0-1 knapsack problems. *Operations Research*, 45, 5, 768–778.

[12] Robert M Nauss. 1976. An efficient algorithm for the 0-1 knapsack problem. *Management Science*, 23, 1, 27–31.

[13] Hazem A. A. Nomer, Khalid Abdulaziz Alnowibet, Ashraf Elsayed, and Ali Wagdy Mohamed. 2020. Neural knapsack: a neural network based solver for the knapsack problem. *IEEE Access*, 8, 224200–224210. DOI: 10.1109/ACCESS.2020.3044005.

[14] Aiying Rong, José Rui Figueira, and Kathrin Klamroth. 2012. Dynamic programming based algorithms for the discounted {0–1} knapsack problem. *Applied Mathematics and Computation*, 218, 12, 6921–6933.

[15] Harvey M. Salkin and Cornelis A. De Kluyver. 1975. The knapsack problem: a survey. *Naval Research Logistics Quarterly*, 22, 1, 127–144. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800220110. DOI: https://doi.org/10.1002/nav.3800220110.

[16] Jingcheng Shen, Kentaro Shigeoka, Fumihiko Ino, and Kenichi Hagihara. 2019. Gpu-based branch-and-bound method to solve large 0-1 knapsack problems with data-centric strategies. *Concurrency and Computation: Practice and Experience*, 31, 4, e4954.

[17] Carnegie Mellon University. 2005. Lecture 10 of the Advanced Approximation Algorithms. Lecture Notes 10. Accessed: 2024-04-22. Carnegie Mellon University. https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf.