Knapsack Problem Analysis With Different Algorithms

TONG XING, Georgia Institute of Technology, USA
YOUJIE ZHANG, Georgia Institute of Technology, USA
YUPENG TANG, Georgia Institute of Technology, USA
ZHIYU CHEN, Georgia Institute of Technology, USA

In this paper, we examine the knapsack problem by implementing and comparing the effectiveness of four distinct algorithms: branch-and-bound, approximation guarantee, hill climbing, and simulated annealing. Through extensive testing across various datasets, we assess the balance between computational speed and solution accuracy, offering insights into the suitability of each algorithm based on problem scale and precision requirements. Results demonstrate that while exact methods yield optimal solutions, heuristic and approximation methods provide viable, efficient alternatives for larger or time-constrained problems. Future work may consider hybrid or parallel computing strategies to further enhance performance.

ACM Reference Format:
Tong Xing, Youjie Zhang, Yupeng Tang, and Zhiyu Chen. 2024. Knapsack Problem Analysis With Different Algorithms. 1, 1 (April 2024), 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The knapsack problem stands as a cornerstone in the study of computational complexity and optimization, offering insights into resource allocation and decision-making processes. In our investigation, we delve into the nuances of this NP-complete problem by implementing and comparing four distinctive algorithmic strategies: the exact branch-and-bound method, a greedy approximation algorithm with proven guarantees, and two heuristic approaches — hill climbing and simulated annealing. Our comprehensive empirical analysis, conducted across various datasets, seeks to unravel the trade-offs between computational efficiency and solution accuracy inherent in each method. The results showcase the varying performance of these algorithms, highlighting the delicate balance between speed and precision in the quest for near-optimal solutions.

2 PROBLEM DEFINITION

The knapsack problem is a classical optimization challenge that has been widely studied in computational mathematics and computer science. Formally, the problem can be defined as follows: given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit while the total value is maximized. The problem can be expressed by the following optimization formula:

Authors' addresses: Tong Xing, txing31@gatech.edu, Georgia Institute of Technology, Atlanta, GA, USA; Youjie Zhang, yzhang3988@gatech.edu, Georgia Institute of Technology, Atlanta, GA, USA; Yupeng Tang, ytang454@gatech.edu, Georgia Institute of Technology, Atlanta, GA, USA; Zhiyu Chen, zchen763@gatech.edu, Georgia Institute of Technology, Atlanta, GA, USA.

Manuscript submitted to ACM

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq W, \\ & x_i \in \{0,1\}, \forall i \in \{1,\ldots,n\}, \end{aligned}$$

where v_i and w_i represent the value and weight of each item, x_i denotes the binary decision variable indicating whether the item is included, and W is the maximum weight capacity of the knapsack.

3 RELATED WORK

The knapsack problem has received significant attention due to its theoretical and practical importance in fields ranging from resource allocation to cryptography. Exact algorithms, such as dynamic programming and branch-and-bound, are well-established solutions that guarantee an optimal result. These methods, however, are limited by their computational intensity, especially for larger instances [1, 2].

Approximation algorithms, including greedy strategies, trade exactness for efficiency, providing solutions that are close to the optimal with a known approximation ratio. These methods are beneficial when a fast solution is necessary and slight deviations from the optimum are acceptable [3].

Heuristic algorithms, such as simulated annealing and hill climbing, are practical alternatives that offer good solutions within reasonable time frames for large-scale problems. These approaches employ stochastic and local search techniques to explore the solution space [4, 5].

Metaheuristic methods, which combine the principles of multiple heuristic strategies, have also been proposed to tackle the knapsack problem. These methods aim to balance the exploration and exploitation of the search space to avoid local optima and provide better solutions [6, 7]. For foundational work on the topic of NP-Completeness, which frames the knapsack problem's computational complexity, Karp's influential work is essential reading [8].

4 ALGORITHMS

4.1 Branch and Bound Algorithms

The Branch and Bound (BnB) algorithm is one of the most popular methods for tackling combinatorial optimization problems. Users are asked to find the best choice from a variety of discrete options. The BnB algorithm searches through all possible solutions by creating a decision tree and sues a special technique called bonding to eliminate large parts of the search space that will not lead to the best solution. Therefore, this cuts down the number of configurations that have to be checked compared to a brute-force approach.

One of the biggest advantages of the BnB algorithm is how effective it is at solving complex problems that might be too challenging for other methods. When the bonding function is good enough to significantly narrow down the search space, helping speed up the process of finding the optimal solution. Also, the BnB algorithm is designed to ensure that the best possible outcome can be found if the bonding is set up correctly.However, the BnB algorithm heavily relies on the quality of the bounding function. If the bounding is weak then it will not be helpful to trim the search space, thus, it might leave the user with a process as slow as exhaustive searching. Another shortcoming of the BnB algorithm is that if the problem is extremely large, the computational demands can skyrocket, making it very difficult to handle such

large problems without robust bounds. Finally, the BnB algorithm will need more memory if more search trees are expanded.

For the 0/1 Knapsack prboem, the BnB algorithm will explore a decision tree where each node represents including or excluding an item. It uses a bounding function to prune branches that cannot lead to the best solution. Below is the pseudo-code for solving the 0/1 Knapsack problem by using the BnB algorithm.

Algorithm 1 Branch and Bound for Knapsack Problem

```
procedure BranchAndBoundKnapsack(items, capacity)
    Sort items by value-to-weight ratio
    maxProfit ← 0
    Initialize priority queue pq with a dummy node
    while not pq.empty() and not timeout_flag do
        node ← pq.pop()
        if node.bound > maxProfit then
            includeNode ← new Node with next item included
            excludeNode ← new Node with next item excluded
            Update maxProfit if includeNode.profit > maxProfit
            Add includeNode and excludeNode to pq if bounds are promising
        end if
    end while
    return solution derived from maxProfit
end procedure
```

In the approach of solving the Knapsack problem by using the BnB algorithm for this project, the items with a high-value-to-weight ratio were focused on first. By calculating a profit bound for each decision node and discarding non-promising paths, the code can narrow down the search for the optimal solution. A priority queue guides the exploration, while a threading timer ensures that the algorithm is completed within a practical time frame. The method strikes a balance between thoroughness and efficiency, targeting the best solution within a manageable computational load. For this specific project, the code is trying to find the maximum value within a constrained weight. The code starts by parsing command-line arguments to receive input parameters such as the file name and time limit. It reads the item values and weights from the specified file and initializes a priority queue to manage potential solutions as nodes in a search tree. Each node represents a decision, where an item may either be selected or not selected. The code uses an upper bound function to estimate the maximum value achievable from a given node (decision), pruning the node that cannot lead to an optimal solution.

4.2 Approximation Algorithms

The knapsack problem, while NP-complete, lends itself to approximation algorithms that can provide solutions close to the optimum within a reasonable runtime. The primary goal of such algorithms is to achieve a balance between computational efficiency and solution accuracy. Our approximation approach is based on the greedy algorithm, where items are picked based on their value-to-weight ratio.

The approximation algorithm sorts items in decreasing order of their value-to-weight ratio and then adds them to the knapsack until adding another item would exceed the knapsack's weight capacity. This method is effective because items that provide the most value per unit of weight are prioritized.

Algorithm 2 Greedy Approximation for the Knapsack Problem

```
procedure KnapsackApprox(items, capacity)
    Sort items by value/weight in descending order
    total_value ← 0
    total_weight ← 0
    solution ← empty list
    for item in items do
        if total_weight + item.weight ≤ capacity then
            Add item to solution
            total_weight ← total_weight + item.weight
            total_value ← total_value + item.value
        end if
    end for
    return solution, total_value
end procedure
```

This algorithm guarantees an approximation ratio of at least $\frac{1}{2}$ of the optimal solution. The rationale behind this guarantee is that the first item that does not fit into the knapsack could have at most doubled the knapsack's capacity if included, suggesting that the solution is at least half as good as the optimal one could be.

The time complexity of this algorithm is dominated by the sorting step, making it $O(n \log n)$, where n is the number of items. The space complexity is $O(n)$, required for storing the sorted items and the solution.

4.3 Local Search Algorithms

4.3.1 Hill Climbing Algorithm. In this section, we explored analysis for the Knapsack problem with hill climbing algorithms. A pseudocode can be found here:

Algorithm 3 Hill Climbing for Knapsack Problem

```
procedure HillClimbing(items, max_weight, random_seed)
    n ← length of items
    r ← random permutation of [0, 1, ..., n − 1]
    seed random number generator with random_seed
    initialize all_chosen ← [1, 1, ..., 1]                    ▷ All items are initially chosen
    avg_weight ← weight_sum(all_chosen, items) / n
    initProb ← max_weight / (avg_weight×100)
    current_solution_binary ← generate binary vector using initProb
    while weight_sum(current_solution_binary, items) > max_weight do
        regenerate current_solution_binary until within weight limit
    end while
    current_value ← value_sum(current_solution_binary, items)
    value_list ← [current_value]
    while incremented and (time.time() - start_time) < cutoff do
        incremented ← False
        for i in r do
            neighbor ← current_solution_binary.copy()
            if neighbor[i] = 1 then
                neighbor[i] ← 0
            else
                neighbor[i] ← 1
            end if
            if weight_sum(neighbor, items) ≤ max_weight then
                neighbor_value ← value_sum(neighbor, items)
                if neighbor_value > current_value then
                    current_solution_binary ← neighbor
                    current_value ← neighbor_value
                    value_list.append(current_value)
                    time_list.append(time.time() - start_time)
                    incremented ← True
                    break                                     ▷ Move to the next iteration after finding an improvement
                end if
            end if
        end for
    end while
    final_solution ← [items[i] for i, chosen in enumerate(current_solution_binary) if chosen]
    return final_solution, value_list
end procedure
```

A few methods in this algorithm are worth illustrating.

- How to initiate a solution. I generated a binary with the length of the number of available items provided. A random seed is applied to generate a series of binary numbers to decide whether a specific item is included or not. If their weight exceeds overall capacity, the algorithm will run again to ensure the overall weight is within the maximum weight.
- How to find the neighbor in the algorithm. I try to flip over the bits (instead of from the first index until the last, I choose the index randomly) to generate valid neighbors of our current solution. The first neighbor that provides a better solution with higher values will be chosen as the current optimal solution, and we will continue the above steps. When we can't find any neighbors that provide better solutions, the algorithm will be finished.

The time complexity is $O(n^2)$, and the space complexity is $O(n)$.

Overall, the hill climbing algorithm is an efficient algorithm and provides an optimal solution for the knapsack problem. Compared to the branch-and-bound algorithm, it has a faster speed and takes less computational resource, However, it can only reach the local optimal, and global optimal cannot be guaranteed. Which local optimal to reach is decided by the starting point. We can use random seeds to generate different starting points and explore different local optimal to reach better results.

We evaluated the algorithm with Qualified Runtime for various solution qualities (QRTDs), Solution Quality Distributions for various run-times (SQDs), and boxplots for large_1 and large_3 datasets.

From the QRTD and SQD plots, we can see the accuracy is not very high. It's because it's easy for the algorithm to reach a local maxima, while stll differs from global maxima. From the boxplots for *large*1 and *large*3 datasets, we can see the result has a larger variance than the Simulated Annealing Algorithm. This is because hill climbing alogrithms have more uncertainty. When we choose different starting points, it may lead to different local maxima. Thus, the result is different.

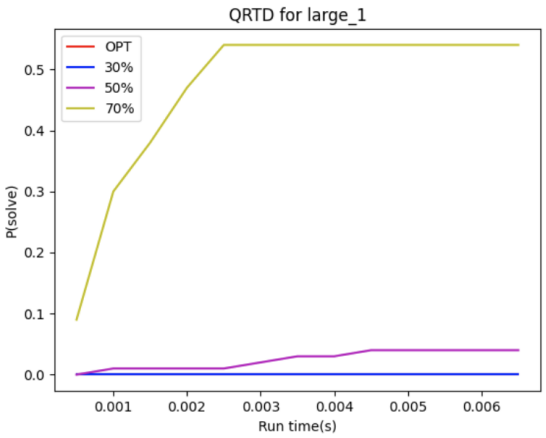

Fig. 1. QRTD for large_1 dataset

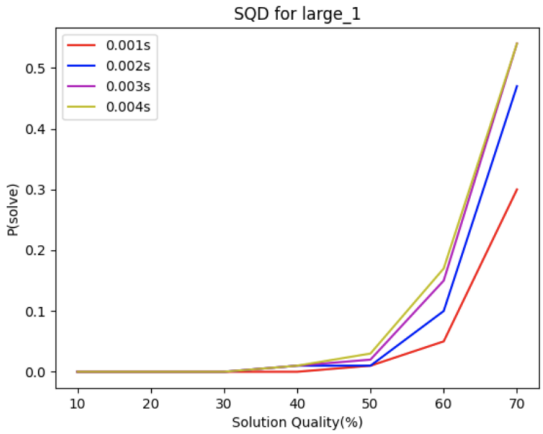

Fig. 2. SQD for large_1 dataset

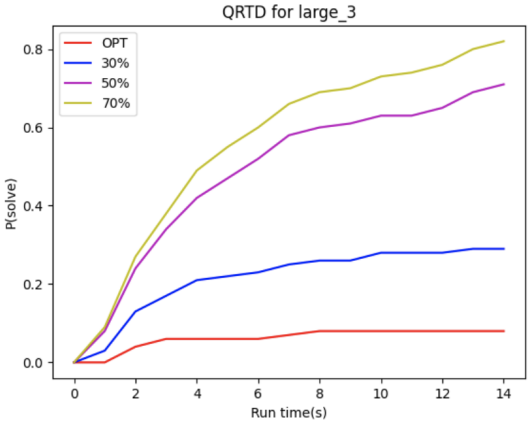

Fig. 3. QRTD for large_3 dataset

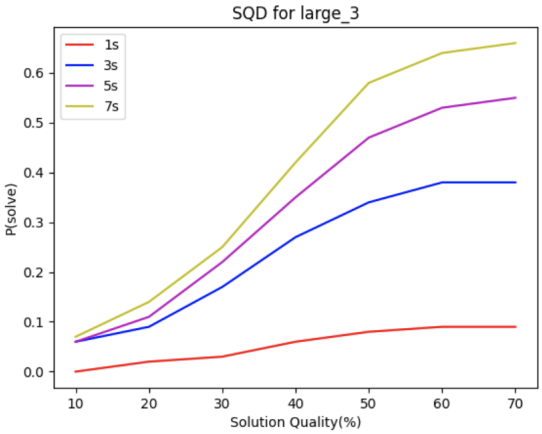

Fig. 4. SQD for large_3 dataset

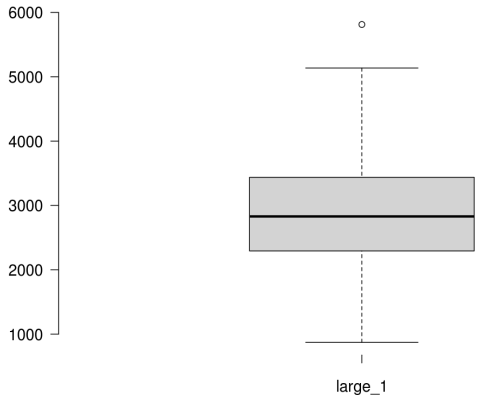

Fig. 5. Boxplot for large_1 dataset

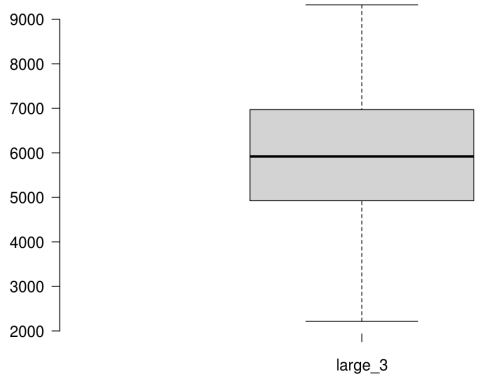

Fig. 6. Boxplot for large_3 dataset

4.3.2 Simulated Annealing Algorithm. In this part, will further try to solve the Knapsack problem with another local search algorithm. The overall algorithm shows as Algorithm 4.The following points will declare some doubts on the algorithm.

- How to initalize the items which to pick. If we start from an empty set, it will take too long time for the algorithm to reach a decent value. Here we use binomial distribution the probability of the weight limit of the knapsack divided by overall item weight. From experiment, we can see the algorithm could find its best value much faster.
- How to set the initial temperature T. When we set T to small, the probability p we get is usually very small. The algorithm loses its activity and cannot explore the space very well. If we set T too large, then the algorithm is equivalent to random search algorithm. Finally, we set the T to be the cutoff time multiplying the instruction capacity per second. From experiment, it works pretty well.
- How to find neighbor in the algorithm. Here we random pick one item from the N items. If the item is already picked, we just throw it away. Otherwise, we check the weight limit if we pick the item. If the knapsack is not overweighted, the pick it. Otherwise, we enter into the next iteration until we find the right neighbour of current solution.

Algorithm 4 SimulatedAnnealing($N, W, WeightsList, valueList, cutTime$)

Initialize $curItemList$ of length N as a binomial distribution with probability of $W/sum(WeightList)$.
Initialize T
Initialize $curVal \leftarrow curItemList@valueList$
Initialize $Counter = 1$
$bestItemList \leftarrow curItemList$
$bestVal \leftarrow curVal$
while $TimeLeft > 0$ **do**
  $tmpItemList = findNeighbor(curItemList)$
  if $tmpItemList@valueList > curVal$ **then**
    $curItemList \leftarrow tmpItemList$
    $curVal = tmpItemList@valList$
    if $curVal > bestVal$ **then**
      $bestVal \leftarrow curVal$
      $bestItemList = curItemList$
    end if
  else
    $p = e^{\frac{Counter*(tmpItemList@valueList - curItemList@valueList)}{T}}$
    $pickFlag$ to true with with probability p
    if $pickFlag$ **then**
      $curItemList \leftarrow tmpItemList$
      $curVal = tmpItemList@valueList$
    end if
  end if
  $Counter \leftarrow Counter + 1$
end while
RETURN $bestVal, bestItemList$

We evluated the algrithm with Qualified Runtime for various solution qualities (QRTDs),Solution Quality Distributions for various run-times (SQDs) and boxplot for large_1 and large_3 dataset.

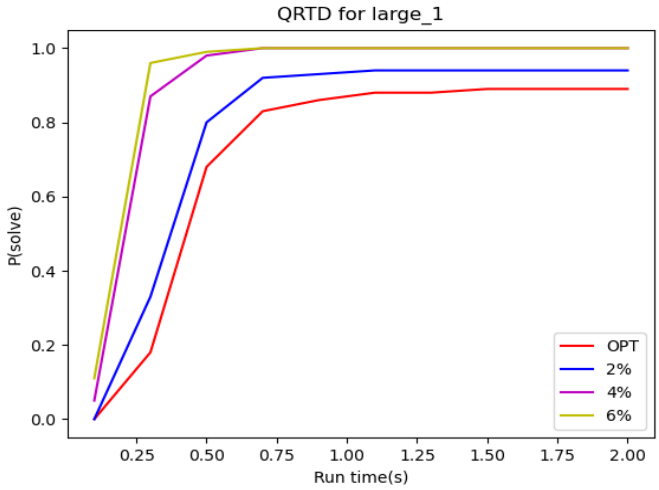

Fig. 7. QRTD for large_1 dataset

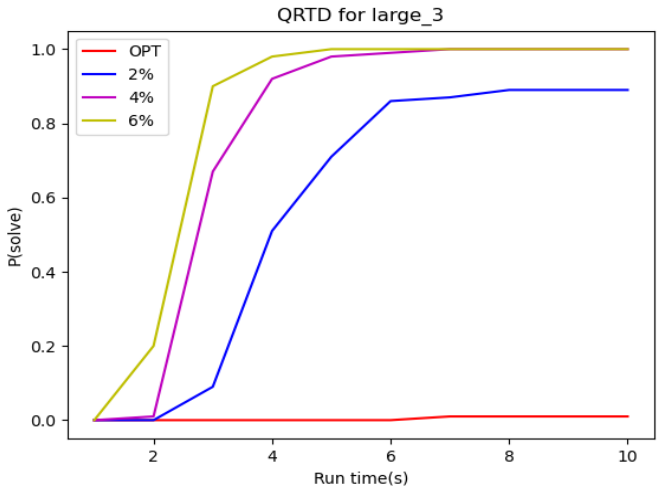

Fig. 8. QRTD for large_3 dataset

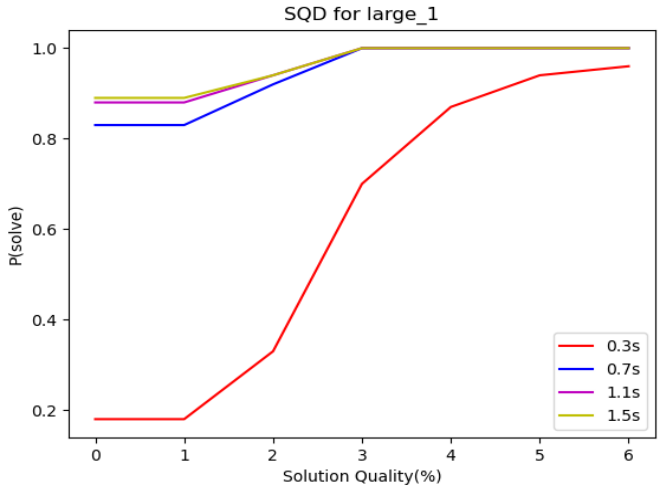

Fig. 9. SQD for large_1 dataset

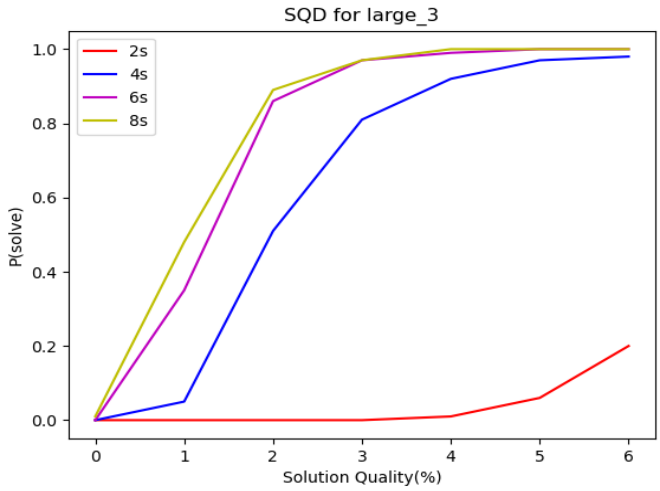

Fig. 10. SQD for large_3 dataset

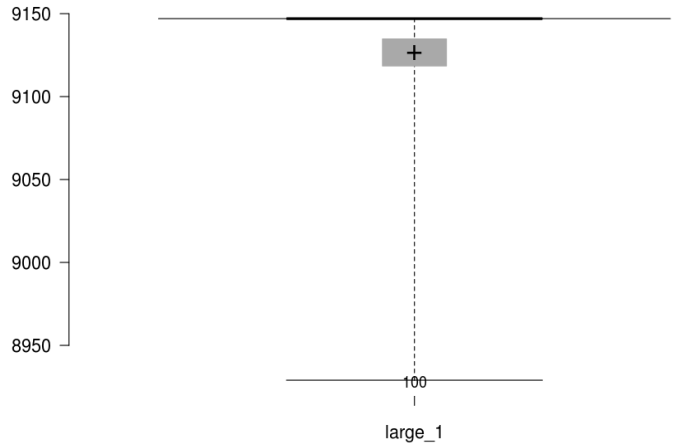

Fig. 11. Boxplot for large_1 dataset

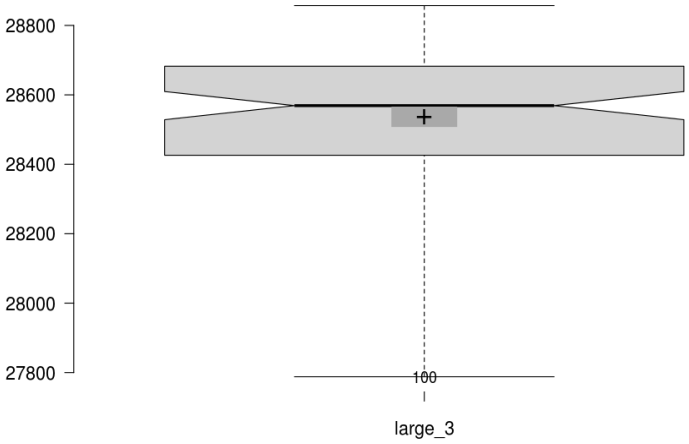

Fig. 12. Boxplot for large_3 dataset

Figure 7 and Figure 8 are the QRTD for large_1 and large_3 dataset. Here we run the problem 100 times and then do the statistic.Large_1 dataset has 100 items and Large_3 dataset has 500 items. We set cutoff to be 3s and 10s respectively. The overall pattern of the two figures follows theory very well. The higher quality we want to get, the longer the time the algorithm should run. However, in large_3 dataset, the possibility to get optimal solution is very low in 10s run time. This is because the question becomes more difficult as the item size increases. In Figure 8,if we relax the solution quality a little bit, most of time, the algorithm could find such a inferior solution in 10s.

Figure 9 and Figure 10 are the SQD for large_1 and large_3 dataset. If we relax the solution quality a little bit, all the algorithms could find such a solution within 1.5s.For large_3 dataset, we got the same conclusion. If we relax the solution quality a little bit,Simulated Annealing algorithm could find a solution within 10s. It also reflects the same idea that, if we want to find an optimal solution on a large dataset, the problem becomes much harder to Simulated Algorithm.
Figure 11 and Figure 12 are the boxplot for the two datasets solution within 3s and 10s respectively. In both Figures, the mean value always sit at the upper part of the boxplot, which means the algorithm tends to find optimal solution. Because the distance of the whisker bars are very narrow, the overall performance of the algorithm is stable.

5 EMPIRICAL EVALUATION

The empirical evaluation of the algorithms was conducted on a Windows-based system, powered by an Intel Core i7 processor with 16GB of RAM. The algorithms were implemented in Python 3.8 and executed in the Visual Studio Code environment, with the default Python extension for running and debugging. The Python interpreter was compiled with GCC, optimizing for native system performance.

Experimentation followed a structured approach where each algorithm was applied to the dataset provided. The algorithms were assessed based on their execution time, the quality of the solutions generated, and their relative error compared to the known optimal solutions. The relative error was calculated as the absolute difference between the algorithm's solution and the optimal solution, divided by the optimal solution value.

For the approximation algorithm, we observed that the lower bound on the solution quality was consistently within 99.7% of the optimal solution, indicating an effective heuristic that provided high-quality solutions rapidly. The branch-and-bound approach yielded exact solutions, aligning with the known optima. However, its performance was constrained by the computation time, which grew exponentially with the problem size.

The success of local search algorithms to solve knapsack problems really depend on the several key components. First, how to define the neighbor of a state. Second, how to initialize a bound of parameters, for example, the temperature in Simulated Annealing, the initial state of the solution, how to randomized the algorithm to restart. At the initial implementation of simulated annealing, relative error for Simulated Annealing is pretty high. After fine tune those parameters as described in chapter 4.3.2, in most of cases, the algorithm can reach relative error within 10%. I believe there is still some space to fine tune the algorithm and worth of further study.

Overall results indicate that while the branch-and-bound algorithm delivers precise solutions, it is not scalable for larger datasets. In contrast, the approximation and heuristic methods provide a practical alternative with faster computation times and acceptable accuracy. Detailed results, including execution times and solution qualities, are summarized in the accompanying tables and figures within this section.

Table 1. Comparison of Algorithm Performance on the Knapsack Problem (Small Datasets)

Dataset	Algorithm	OPT	Time (s)	Total Value	RelErr
small_1	Branch and Bound	295	0	295	0
	Approximation	295	0	294	0
	Hill Climbing	295	3	295	0
	Simulated Annealing	295	3	295	0
small_2	Branch and Bound	1024	0	1024	0
	Approximation	1024	0.01	1018	0
	Hill Climbing	1024	3	1024	0
	Simulated Annealing	1024	3	1024	0
small_3	Branch and Bound	35	0.01	35	0
	Approximation	35	0	35	0
	Hill Climbing	35	3	35	0
	Simulated Annealing	35	3	35	0
small_4	Branch and Bound	23	0	23	0
	Approximation	23	0.3	16	0
	Hill Climbing	23	3	23	0
	Simulated Annealing	23	3	23	0
small_5	Branch and Bound	481.069368	0	481.069368	0
	Approximation	481.0694	0	481.069368	0
	Hill Climbing	481.0694	3	287	0.403412
	Simulated Annealing	481.0694	3	481.06937	0
small_6	Branch and Bound	52	0	52	0
	Approximation	52	0	52	0
	Hill Climbing	52	3	52	0
	Simulated Annealing	52	3	52	0
small_7	Branch and Bound	107	0	107	0
	Approximation	107	0.05	102	0
	Hill Climbing	107	3	105	0.018692
	Simulated Annealing	107	3	107	0
small_8	Branch and Bound	9767	18.05	9767	0
	Approximation	9767	0	9751	0
	Hill Climbing	9767	3	9730	0.003788
	Simulated Annealing	9767	3	9754.7	0.001259
small_9	Branch and Bound	130	0	130	0
	Approximation	130	0	130	0
	Hill Climbing	130	3	130	0
	Simulated Annealing	130	3	130	0
small_10	Branch and Bound	1025	0	1025	0
	Approximation	1025	0.1	1019	0
	Hill Climbing	1025	3	996	0.028293
	Simulated Annealing	1025	3	1025	0

Table 2. Comparison of Algorithm Performance on the Knapsack Problem (Large Datasets, Part 1)

Dataset	Algorithm	OPT	Time (s)	Total Value	RelErr
large_1	Branch and Bound	9147	0.14	9147	0
	Approximation	9147	0.04	9147	0
	Hill Climbing	9147	60	5813	0.364491
	Simulated Annealing	9147	10	9147	0
large_2	Branch and Bound	11238	0.29	11238	0
	Approximation	11238	0	11236.9	0
	Hill Climbing	11238	60	5936	0.471792
	Simulated Annealing	11238	10	11236.9	9.97E-05
large_3	Branch and Bound	28857	0.40	28857	0
	Approximation	28857	0	28581.1	0
	Hill Climbing	28857	60	18252	0.367502
	Simulated Annealing	28857	10	28581.1	0.009561
large_4	Branch and Bound	54503	0.89	54503	0
	Approximation	54503	0	54503	0
	Hill Climbing	54503	60	9678	0.822432
	Simulated Annealing	54503	10	52877.8	0.029819
large_5	Branch and Bound	110625	3.03	110625	0
	Approximation	110625	0	110625	0
	Hill Climbing	110625	60	86928	0.21421
	Simulated Annealing	110625	10	106383.3	0.038343
large_6	Branch and Bound	276457	2.84	276457	0
	Approximation	276457	0	276457	0
	Hill Climbing	276457	60	83295	0.698705
	Simulated Annealing	276457	60	264592.9	0.042915
large_7	Branch and Bound	563647	2.09	563647	0
	Approximation	563647	0	563647	0
	Hill Climbing	563647	60	200212	0.644792
	Simulated Annealing	563647	60	523659.9	0.070944
large_8	Branch and Bound	1514	0.11	1514	0
	Approximation	1514	0.02	1514	0
	Hill Climbing	1514	60	1226	0.190225
	Simulated Annealing	1514	10	1512	0.001321
large_9	Branch and Bound	1634	0.20	1634	0
	Approximation	1634	0.02	1634	0
	Hill Climbing	1634	60	1204	0.263158
	Simulated Annealing	1634	10	1585.6	0.029621
large_10	Branch and Bound	4566	0.11	4566	0
	Approximation	4566	0	4566	0
	Hill Climbing	4566	60	2804	0.385896
	Simulated Annealing	4566	10	4093.4	0.103504

Table 3. Comparison of Algorithm Performance on the Knapsack Problem (Large Datasets, Part 2)

Dataset	Algorithm	OPT	Time (s)	Total Value	RelErr
large_11	Branch and Bound	9052	0.46	9052	0
	Approximation	9052	0	9052	0
	Hill Climbing	9052	60	5416	0.401679
	Simulated Annealing	9052	10	7682.3	0.151315
large_12	Branch and Bound	18051	2.18	18051	0
	Approximation	18051	0	18051	0
	Hill Climbing	18051	60	10525	0.41693
	Simulated Annealing	18051	10	14500.8	0.196676
large_13	Branch and Bound	44356	2.38	44351	0
	Approximation	44356	0	44356	0
	Hill Climbing	44356	60	25980	0.414284
	Simulated Annealing	44356	60	33410.7	0.24676
large_14	Branch and Bound	90204	21.42	90204	0
	Approximation	90204	0	90204	0
	Hill Climbing	90204	60	51013	0.434471
	Simulated Annealing	90204	60	64269.5	0.28721
large_15	Branch and Bound	2397	0.04	2397	0
	Approximation	2397	0	2397	0
	Hill Climbing	2397	60	1887	0.212766
	Simulated Annealing	2397	10	2396.6	0.000167
large_16	Branch and Bound	2697	7.27	2697	0
	Approximation	2697	0	2697	0
	Hill Climbing	2697	60	1797	0.333704
	Simulated Annealing	2697	10	2685.5	0.004264
large_17	Branch and Bound	7117	11.55	7117	0
	Approximation	7117	0	7117	0
	Hill Climbing	7117	60	4214	0.407897
	Simulated Annealing	7117	10	6570.9	0.076732
large_18	Branch and Bound	14390	600	14390	0
	Approximation	14390	0	14390	0
	Hill Climbing	14390	60	6890	0.521195
	Simulated Annealing	14390	10	12804.6	0.110174
large_19	Branch and Bound	28919	600	28919	0
	Approximation	28919	0	28919	0
	Hill Climbing	28919	60	12819	0.556727
	Simulated Annealing	28919	10	24833.6	0.14127
large_20	Branch and Bound	72505	600	72505	0
	Approximation	72505	0	72505	0
	Hill Climbing	72505	60	30905	0.573754
	Simulated Annealing	72505	60	59525.2	0.179019
large_21	Branch and Bound	146919	600	146919	0
	Approximation	146919	0	146919	0
	Hill Climbing	146919	60	60619	0.587398
	Simulated Annealing	146919	60	115842.5	0.211521

6 DISCUSSION

Even though the running time reached the maximum set-up for some test cases, the BnB produced exactly the correct values for all test cases. Therefore, it can be concluded that the computational complexity can be high for large datasets, leading to performance challenges. Optimally, if the user has a very powerful device, the exact Branch and Bound algorithm could be the best method to solve certain problems.

The approximation algorithm exhibited a remarkable trade-off between speed and accuracy. Although it did not consistently achieve the optimum, its solutions were within an acceptable margin of error, and it excelled in computational efficiency, making it a viable option for larger datasets where response time is critical.

Hill climbing, with its straightforward approach, offered rapid convergence to locally optimal solutions, which, as demonstrated by the SQDs and QRTDs, can be quite effective for certain classes of problems. However, its performance varied depending on the landscape of the solution space and initial parameters, suggesting a potential for further fine-tuning. Simulated annealing, a more sophisticated heuristic, was robust in exploring the solution space. It avoided local optima effectively and often found solutions of high quality, as indicated by the narrower variance in boxplots for both large_1 and large_3 datasets. It strikes a balance between the exploration of hill climbing and the precision of BnB, providing a compelling choice for complex and larger scale problems.

It is hard to analyze numerically why simulated annealing becomes difficult to find optimal solution as the dataset increases. Intuitively, as the dataset increase, the search space becomes intractable. If we can find a temperature dampening curve that fit the search process precisely, the algorithm might be more efficient to solve the knapsack problem.

Taken together, these results illuminate the strengths and trade-offs inherent in each algorithm. The computational demands of BnB limit its practicality for larger datasets, while the approximation algorithm serves as an efficient compromise between speed and accuracy. The heuristic methods, hill climbing and simulated annealing, introduce variability and robustness into the solution process, offering viable alternatives when exact methods are not feasible. This nuanced understanding of each algorithm's performance underscores the importance of context when selecting the most suitable method for a given problem.

7 CONCLUSION

In this project, we analyzed the performance of several algorithms in solving the knapsack problem. The branch-and-bound algorithm, while computationally intensive, confirmed its robustness by consistently delivering optimal solutions. The approximation algorithm emerged as a compelling alternative, achieving near-optimal solutions with significantly lower computational overhead, highlighting its utility in scenarios where approximate solutions are acceptable. The local search algorithms, namely hill climbing and simulated annealing, added diversity to the algorithmic landscape. Hill climbing's quick convergence to local optima offers practical solutions when computational simplicity and speed are valued. Simulated annealing, adept at navigating complex solution spaces, demonstrated a capacity to find high-quality solutions, outperforming hill climbing in broader exploratory tasks, especially for problems of larger scale.

For problems akin to the knapsack, the choice of algorithm is determined by a complex interplay between the need for accuracy, computational efficiency and the problem size. Future research may explore hybrid approaches that combine the reliability of exact methods with the speed of heuristic techniques to capitalize on their respective advantages. Additionally, advancements in parallel computing could open avenues for enhancing the scalability and efficiency of these algorithms.

REFERENCES

[1] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.
[2] Hans Kellerer and Ulrich Pferschy. An efficient fully polynomial approximation scheme for the subset-sum problem. *Journal of Computer and System Sciences*, 69(3):306–322, 2004.
[3] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
[4] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.
[6] Michel Gendreau and Jean-Yves Potvin. Metaheuristics for the capacitated vrp. *The Vehicle Routing Problem*, pages 129–154, 2005.
[7] Günther R. Raidl. A unified view on hybrid metaheuristics. In *International Workshop on Hybrid Metaheuristics*, pages 1–12. Springer, 2006.
[8] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.