Solving the intractable problem of Knapsack with non-exhaustive search, approximation, and heuristic algorithms

JUAN ANTONIO ROBLEDO LARA, Georgia Institute of Technology, US
RISHI BANERJEE, Georgia Institute of Technology, US
GRAHAM P VAITH, Georgia Institute of Technology, US
KIRAN NAZARALI, Georgia Institute of Technology, US

ACM Reference Format:
Juan Antonio Robledo Lara, Rishi Banerjee, Graham P Vaith, and Kiran Nazarali. 2024. Solving the intractable problem of Knapsack with non-exhaustive search, approximation, and heuristic algorithms. 1, 1 (May 2024), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Knapsack problem has been proven to be an NP-Complete problem. Cargo loading and capital budgeting are real-life applications of the Knapsack problem where we are trying to maximize the total cargo or total payoff while satisfying some constraint. For the Knapsack problem, we are given a list of items along with their value and weight as well as a weight constraint. Our goal is to select a subset of items such that we maximize the total value of all the items while satisfying the weight constraint.

The original dynamic programming approach to solve Knapsack for an exact solution can be infeasible, especially for large instances. Therefore, our approach to solve Knapsack was to implement three different types of algorithms that instead arrive at a potentially suboptimal solution in a reasonable amount of the time. Though the solution may be suboptimal, we strive to arrive at a solution that has good enough quality for our purposes in this reasonable amount of time.

The three types of algorithms that we will explore are an exact algorithm involving Branch-and-Bound (BnB), greedy algorithm with approximation guarantees to the optimal solution, simulated annealing (SA), and hill climbing (HC). Our hope is to use these algorithms to obtain a solution with high enough quality in a reasonable amount of time.

We distributed the coding work for four algorithms among our four group members to generate results for a set of traces for knapsack. Overall, the algorithms reveal different tradeoffs between time and accuracy of the solution in comparison to the optimal solution.

2 PROBLEM DEFINITION

In the knapsack problem, we are given a set of N items, each with a value v_i and a weight w_i. Both v_i and w_i are integers. The goal is to select a subset of these items such that the total weight does not exceed a given capacity W, which is a positive integer.

Formally, we seek to maximize the total value of the selected items:

$$\text{Maximize} \quad \sum_{i=1}^{N} v_i x_i$$

subject to the weight constraint:

$$\sum_{i=1}^{N} w_i x_i \leq W,$$

where $x_i \in \{0, 1\}$ indicates whether item i is included ($x_i = 1$) or not included ($x_i = 0$) in the knapsack.

For the purposes of our algorithms, we primarily consider this binary selection framework. However, there are variations where x_i can take on any non-negative integer value, representing the number of instances of item i included in the knapsack. These variations introduce different complexities into the problem-solving approach.

3 RELATED WORK

The knapsack problem encompasses a wide range of variants, including bounded and unbounded knapsack, subset-sum, change-making, and multiple knapsack problems. Additionally, related problems such as bin-packing and the generalized assignment problem can be seen as extensions of the knapsack problem [4].

Branch-and-Bound algorithms that solve these problems exactly often start by sorting items by decreasing weight. This approach has been shown to minimize the number of iterations required [4]. In variations of the knapsack problem with a conflict graph, one effective upper bound strategy for Branch-and-Bound involves ordering items by increasing value/weight ratios and calculating the upper bound by inserting items along with a fractional copy of an item, if possible. Another strategy involves using the capacitated weighted clique cover bound, which approximates the weighted clique cover algorithm [3]. A new dynamic programming version of Branch-and-Bound, called Balsub, now outperforms other approaches on standard benchmark instances [4].

Approximation algorithms, which provide theoretical guarantees of achieving a certain approximation ratio, also play a significant role. For instance, the standard greedy algorithm for fractional knapsack offers an approximation ratio of 0.5. A multiple-pass variation of this algorithm has achieved an improved approximation ratio of 0.75, and other greedy variations have pushed this ratio to approximately 0.8 [4].

Furthermore, genetic metaheuristic algorithms have been extensively used to address the knapsack problem, particularly valuable

Authors' addresses: Juan Antonio Robledo Lara, Georgia Institute of Technology, Atlanta, US; Rishi Banerjee, Georgia Institute of Technology, Atlanta, US; Graham P Vaith, Georgia Institute of Technology, Atlanta, US; Kiran Nazarali, Georgia Institute of Technology, Atlanta, US.

when the value/weight ratios are substantial for large instances. Examples include hybrid grouping GA, weighted coding GA, undominated grouping GA, and representation-RSGA. Additionally, local search algorithms, such as simple iterative or second iterative local search, employ various neighborhood strategies like Replace-One-By-One, Replace-Two-By-One, and Replace-One-By-Two, demonstrating their effectiveness in solving knapsack problems [2].

4 ALGORITHMS

4.1 Branch-and-Bound Implementation

In our Branch-and-Bound implementation for the knapsack problem, we transformed the objective of maximizing total value into minimizing negative total value. This transformation is performed as follows:

(1) Multiplying the values by -1 to invert the problem's optimization direction.
(2) Selecting the node with the smallest cost (or lower bound) from the frontier as the "most promising node".
(3) Eliminating nodes if their cost (or lower bound) exceeds the current best solution's upper bound, thereby not adding them to the frontier.

After the algorithm completes, we convert the problem back to its original maximization form by multiplying the minimum negative total value (which is also the upper bound of the current best solution) by -1.

As a lower bound, we calculate the negative total value possible by satisfying the current remaining weight using items that have not been considered yet. To ensure that this is a lower bound, we can also add a fraction of the last item with its negative fractional value. We can consider the smallest lower bound as the most "promising" configuration.

Since this implementation uses a binary tree, there are only 2 possible children configurations after the Expand() step: including the new item or not including the new item.

The pseudocode used for this minimization version of the Branch-and-Bound algorithm was introduced in Lecture 20, as illustrated in Figure 1.

```
Branch-and-Bound(P) // Input: minimization problem P
01 F <- {(∅,P)}   // Frontier set of configurations
02 B <- (+∞ , (∅,P))   // Best cost and solution
03 while F not empty do
04   Choose (X,Y) in  F – the most "promising" configuration
05   Expand (X,Y), by making a choice(es)
06   Let (X1, Y1), (X2, Y2), ..., (Xk, Yk) be new configurations
07   for each new configuration (Xi,Yi) do
08     "Check" (Xi,Yi)
09     if "solution found" then   (a complete solution, not necessarily the optimal solution)
10       if cost(Xi) < B cost then // update upper bound
11         B <- (cost(Xi),(Xi,Yi))
12     if not "dead end" then
13       if lb(Xi) < B cost then //
14         F <- F U {(Xi,Yi)}   // else prune by lb
15 return B
```

Fig. 1. Pseudocode for the Branch-and-Bound Minimization Algorithm

This implementation strategy was inspired by the structured approach of using a Branch-and-Bound node as a class, with the algorithm encapsulated within, similar to methods described on GeeksforGeeks [1].

The time and space complexity of this approach remains $O(2^n)$, where n is the number of items, reflecting the exponential growth of the search space typically explored by Branch-and-Bound algorithms.

4.2 Simulated Annealing Implementation

For our first local search algorithm, we chose Simulated Annealing (SA), a probabilistic technique that approximates the global optimum of a given function.

4.2.1 *Algorithm Components.*

- **Initialization:** The algorithm initializes by reading the problem data from a file, setting up parameters, and deciding on an initial solution state which can be random or based on a greedy approach. The choice to include a greedy approach for initialization was influenced by discussions on Piazza.
- **Temperature Management:** A temperature parameter is utilized to control the acceptance probability of worse solutions, facilitating thorough exploration of the solution space. We have implemented an optional heating mechanism [6] that dynamically determines the initial temperature based on the desired initial acceptance probability. This mechanism adjusts the starting temperature so that the probability of accepting worse solutions aligns with a target threshold, facilitating an efficient start.
- **Local Search via Flipping:** The core of the local search involves flipping the inclusion state of items in the knapsack randomly, allowing the exploration of nearby solutions.
- **Adaptive Flips:** The algorithm adapts the number of flips based on the acceptance rate of new states, dynamically balancing between exploration and exploitation. During experimentation, we noticed that flipping more elements yielded better results more quickly in smaller instances. Users can choose the number of initial flips.
- **Cooling Schedule:** The temperature is gradually reduced according to a predefined geometric cooling rate [6], decreasing the likelihood of accepting worse solutions as the process progresses.
- **Restart Mechanism:** We implemented an optional restart mechanism inspired by content reviewed during lectures. Our restart mechanism performs the restart twice: During the first pass, we start with a random initialization; then the second pass uses the best state found so far in the first pass as the initial state. Finally, in the third pass, we switch to a greedy initialization and run a third pass to see if we can improve on what has been obtained so far.

4.2.2 *Pseudo-code.*

4.2.3 *Time and Space Complexities.* The time complexity is primarily $O(\text{max\_iter} * n * \text{restart\_iter})$ due to the iterations over solution evaluations, where *max_iter* is the maximum number of iterations, n is the number of items, and *restart_iter* represents the number of

Algorithm 1 Simulated Annealing for the Knapsack Problem

Input: Input data including items and their values and weights
Output: Best subset of items maximizing value within weight limit
Initialize parameters and read input data
if with_greedy_init **then**
    $curr\_state \leftarrow$ generate_greedy_feasible_initial_state()
else
    $curr\_state \leftarrow$ initialize random state
end if
Set initial temperature T
for $i = 1$ to $max\_iterations$ **do**
    for $j = 1$ to $restart\_iterations$ **do**
        $candidate\_state \leftarrow$ flip_random_elements($curr\_state$)
        Evaluate δ_value between $candidate\_state$ and $curr\_state$
        if AcceptanceCriteria(δ_value, T) **then**
            $curr\_state \leftarrow candidate\_state$
        end if
        Update best found solution if candidate is better
        Adjust flips based on acceptance rate
        Reduce temperature T according to $cooling\_rate$
        Break if time limit exceeded or temperature below threshold
    end for
    Optionally restart from best state or greedy state
end for
Output best solution found

restarts. Space complexity is $O(n)$, governed by the storage of item states and properties.

4.2.4 *Strengths and Weaknesses.*

Strengths.

- Flexibility and adaptability in escaping local optima.
- Effective exploration and exploitation balance through adaptive local search mechanisms.
- The running time needed to obtain decent results is not very high.

Weaknesses.

- High sensitivity to the settings of its parameters like the initial temperature cooling rate, and the initial state.
- The performance and the parameters depends a lot in the input data.

4.2.5 Configuration and Tuning. In Table 1, we present the parameters we tested in our SA algorithm. We found that the automatic heating mechanism did not significantly improve the performance of the algorithm; it was more effective to set a fixed initial temperature of 1000. Moreover, the initial flips were set to 3 for large instances and 5 for small instances with the adaptive flips mechanism activated. We observed that large instances often reached the weight limit with very few objects in the knapsack, making it more efficient to have fewer flips to avoid spending excessive time exploring unfeasible solution spaces. The optimal cooling rate was found to be 0.995, which causes a slower decrease in temperature, allowing for more iterations and, consequently, better results. Additionally, using greedy initialization consistently provided a good approximation of the optimal value. However, the algorithm often became stuck in local minima, leading to no improvement over subsequent iterations. To address this, we developed a restart mechanism that alternates between a random start and a greedy start, thus increasing the chances of escaping local minima. In summary, the best parameters we used for the evaluating the algorithm are shown in Table 2.

Table 1. Tuning Parameters for Simulated Annealing

Parameter	Possible Settings
Auto Heating	On, Off
Initial Flips	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Adaptive Flips	On, Off
Cooling Rate	0.80, 0.90, 0.995
Restart	On, Off
Greedy Initialization	On, Off
Max Iterations	10,000; 20,000; 80,000

Table 2. Used Parameters for Simulated Annealing

Parameter	Final Settings
Auto Heating	Off
Initial Flips	3 for large, 5 for small
Adaptive Flips	On
Cooling Rate	0.995
Restart	On
Greedy Initialization	Off but used in restart
Max Iterations	80,000 per restart pass

4.3 Hill Climbing Implementation

In order to implement the hill climbing algorithm, the model needs to, at a node, check of any of the direct neighbors of the current state in the search space, and change to another node if there is any node that causes an increase in the overall value that is either greater than or equal to the current value. If the function found is the maximum value between it and all of its neighbors, then the configuration that is found and its corresponding value are then returned. The format of each state is that, for each item in item set N, is a value in $\{0, 1\}^{|N|}$, since each item can either be included or not in any configuration. The neighbor states could be found, for each of the $|N|$ values, flipping a single one of the items from included to excluded or vice versa.

The pseudocode for the hill climbing algorithm is shown below:

This code is dependent on the random initialization of the starting states. For large item set values, it is possible for the number of viable nodes (i.e. have a total weight less than that of the max weight) to be much less the total number of states for the number of items. As a result, the algorithm that was used to produce a viable starting

Algorithm 2 Hill Climbing Algorithm for the Knapsack Problem

Input: N (set of items), B (knapsack capacity)
Output: Best subset of items within capacity
Step 1: Randomly initialize a starting state s in $\{0,1\}^{|N|}$ with total weight less than B
Step 2: Greedily choose the best next state, returning if there are none better
while total time elapsed < cutoff time **do**
    Get set S of next nodes that could be moved to (i.e., does not exceed max weight)
    if there is no option in S with a higher value than s **then**
        return s, total value of s
    end if
    for each option in S **do**
        if total value of option > total value of s **then**
            $s \leftarrow$ option
        end if
    end for
end while

state used 50 percent division between inclusion and exclusion as a starting point, and for each round where a viable starting node was not found, discounted the probability of including an item by a factor of 0.99.

4.4 Approximation Implementation

The approximation algorithm implemented was a modified version of the greedy algorithm. This algorithm sorts items based on their profit-to-size ratio and selects them greedily until the knapsack's capacity is reached. Additionally, it compares the total profit of the selected items to the most profitable single item that can fit in the knapsack, choosing the better of the two solutions.

4.4.1 Approximation Guarantee and Complexity. The ModifiedGreedy algorithm guarantees at least half the optimal profit, making it a 2-approximation algorithm for the Knapsack problem [5]. Its time complexity is $O(n \log n)$ due to sorting, and it operates in linear space, $O(n)$, with respect to the number of items.

4.4.2 Strengths and Weaknesses.

Strengths:

- *Simplicity*: The algorithm is straightforward and easy to implement.
- *Efficiency*: It provides a quick solution, which is useful when a near-optimal solution is sufficient.
- *Approximation Guarantee*: It ensures a profit that is at least half of the optimal.

Weaknesses:

- *Non-optimality*: It does not always yield the optimal solution.
- *Ratio-dependent*: Its performance can degrade with certain distributions of profit-to-size ratios.

Algorithm 3 Modified Greedy Algorithm for the Knapsack Problem

Input: N (set of items), B (knapsack capacity)
Output: Subset of items maximizing profit without exceeding capacity
Each item i has a size s_i and a profit p_i
Step 1: Sort items in N by $\frac{p_i}{s_i}$ in descending order
Step 2: Greedily add items to the knapsack
Initialize $A = \emptyset$ ▷ A will hold the subset of chosen items
Initialize $currentSize = 0$
Initialize $maxProfitItem = N[1]$ ▷ Assume N is sorted
for each item i in N **do**
    if $currentSize + s_i \leq B$ **then**
        Add i to A
        $currentSize += s_i$
    else
        if $p_i > p(maxProfitItem)$ **then**
            $maxProfitItem = i$
        end if
    end if
end for
Step 3: Compare the greedy solution with the most profitable single item
if $profit(A) > p(maxProfitItem)$ **then**
    return A
else
    return $\{maxProfitItem\}$ ▷ Return the set containing only the most profitable item
end if

5 EMPIRICAL EVALUATION

In this section, we present the results obtained during the evaluation of each algorithm using different input files. In Table 3, we present the specifications of the computers that were used to evaluate each algorithm.

Table 3. System specifications for each algorithm

Algorithm	Processor	RAM	Language
BnB	AMD Ryzen 5 3500	16 GB	Python
SA	Intel Core i7-1260P	16 GB	Python
HC	Intel Core i7-1068NG7	16 GB	Python
Approx	AMD Ryzen 7 5800X	32 GB	Python

5.1 Branch-and-Bound Evaluation

All Branch-and-Bound instances were run with a time cutoff of 300 seconds. The results were obtained from running the algorithms and returning the best quality solution or upper bound when the algorithm terminates. The algorithm can either terminate from not having any more nodes to explore or from exceeding the time cutoff. We were able to calculate relative error from the best quality solution returned form Branch-and-Bound and recorded these results in the comprehensive table.

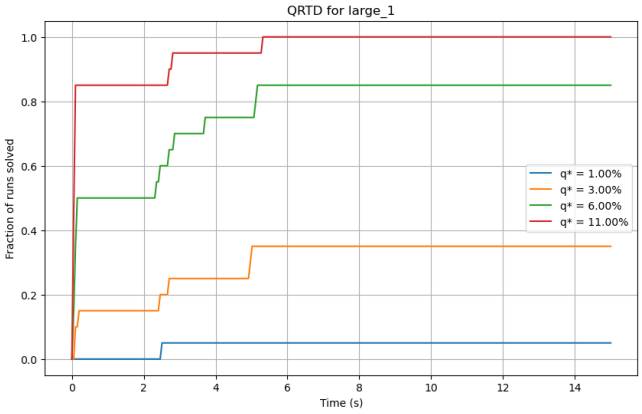

Fig. 2. SA QRTD Graph for large_1 Instance

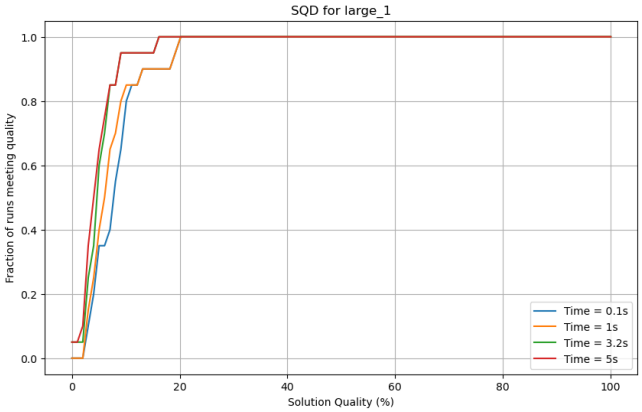

Fig. 3. SA SQD Graph for large_1 Instance

5.2 Simulated Annealing Evaluation

The results for this algorithm presented in Table 5 were derived from running the simulated annealing algorithm 10 times per instance with various random seeds, using the parameters outlined in Table 2.

In addition to the comprehensive table, we conducted a more detailed analysis of two large instances, large_1 and large_3. For this analysis, the algorithm was run 20 times for each instance, generating trace files that facilitated the creation of qualified run-time distribution (QRTD) graphs, solution quality distribution (SQD) graphs, and box plots for the runtime of each instance.

5.3 Hill Climbing Evaluation

For this analysis, the algorithm was run 100 times for the large_1 and large_3, generating trace files that facilitated the creation of qualified run-time distribution (QRTD) graphs, solution quality distribution (SQD) graphs, and box plots for the runtime of each instance. In order to generate the data for the comprehensive table, the data was run ten times for each input file with 10 different random seeds.

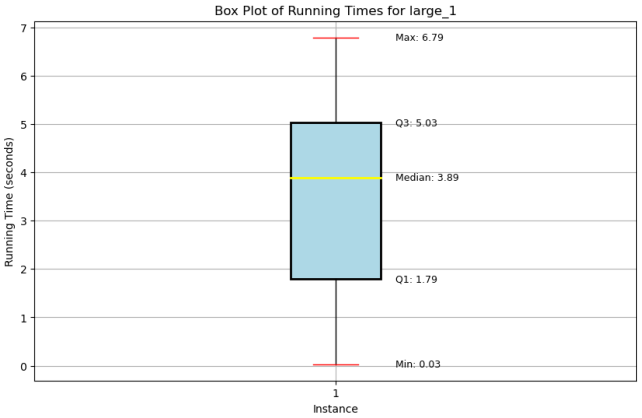

Fig. 4. SA Runtime Boxplot for large_1 Instance

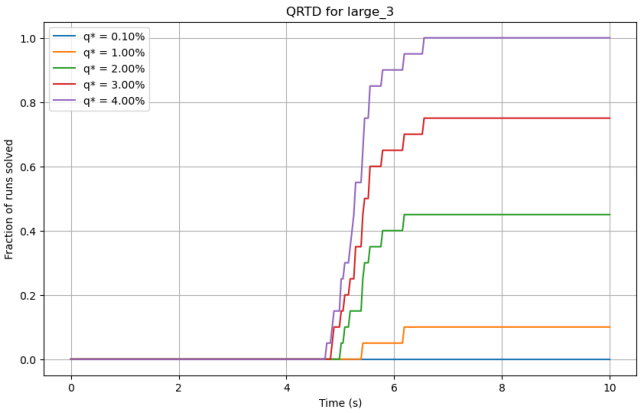

Fig. 5. SA QRTD Graph for large_3 Instance

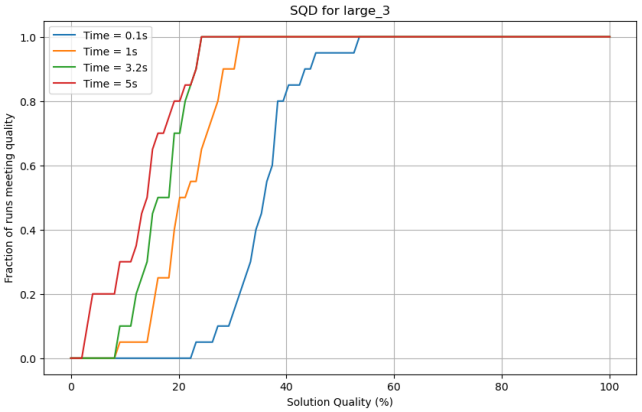

Fig. 6. SA SQD Graph for large_3 Instance

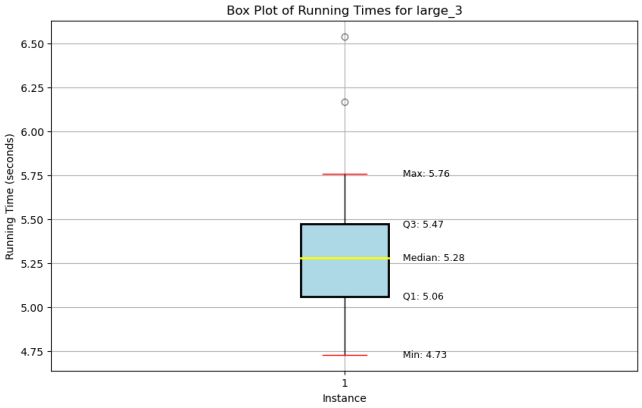

Fig. 7. SA Runtime Boxplot for large_3 Instance

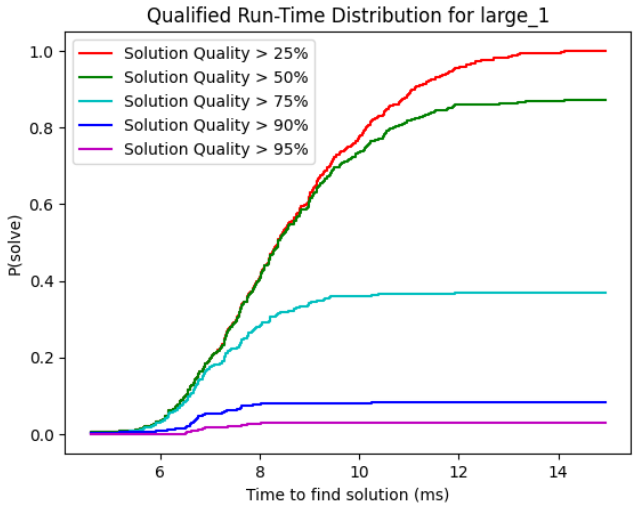

Fig. 8. HC QRTD Graph for large_1 Instance

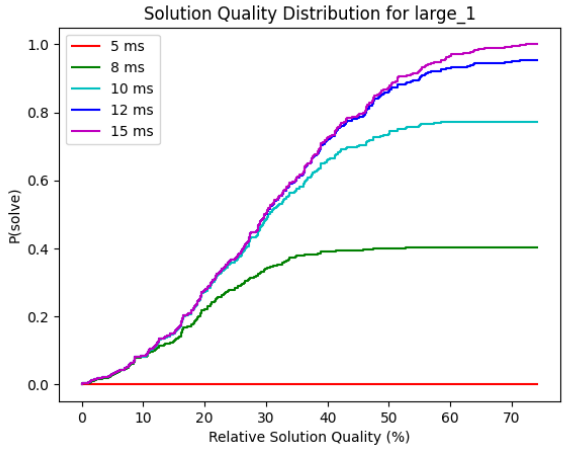

Fig. 9. HC SQD Graph for large_1 Instance

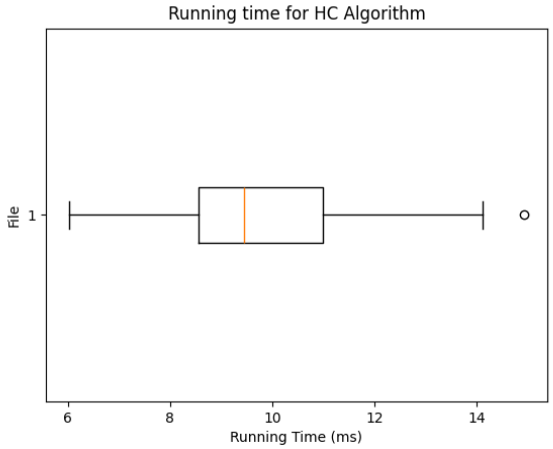

Fig. 10. HC Runtime Boxplot for large_1 Instance

The values for this data were parsed from the data in the trace and solution files.

5.4 Approximation Evaluation

The results for the Approximation algorithm can be seen in Table 4, where the results were obtained from running the Modified Greedy algorithm on the provided datasets. A cutoff time of 300s was used for each one, but this was never close to being reached. The algorithm was run on all available datasets only once, and the results were sent to a csv.

6 DISCUSSION

6.1 Branch-and-Bound Discussion

Branch-and-Bound explores the entire state space, exhibiting exponential runtime in the worst case without pruning. Its main strength lies in its ability to provide an exact solution for many instances, resulting in a relative error of zero when the algorithm runs to completion. However, its primary weakness is that if Branch-and-Bound cannot run to completion, the relative error on several large-scale instances becomes significantly high.

For small instances, the relative error is less than 1%, demonstrating high accuracy as Branch-and-Bound can perform an exact search within the time cutoff, yielding the correct or nearly correct answer.

Conversely, for large instances such as large 7, large 14, and large 21, the relative error exceeds 50%, and all exceed the 300-second runtime limit. This underscores that as input size increases, Branch-and-Bound's runtime grows exponentially, making it less feasible and accurate with its current implementation.

Table 4. Comprehensive Table for Branch and Bound and the Approximation Algorithms

Dataset	Input Size	Branch and Bound			Approximation		
		Time (s)	Value	RelErr	Time (s)	Value	RelErr
small_1	10	0.0	295	0.0	0.0	294	0.003
small_2	20	0.016	1024	0.0	0.0	1018	0.006
small_3	4	0.0	35	0.0	0.0	35	0.000
small_4	4	0.0	23	0.0	0.0	16	0.304
small_5	15	0.0	481	6.65e-8	0.0	481.069	6.65e-08
small_6	10	0.0	52	0.0	0.0	52	0.0
small_7	7	0.0	107	0.0	0.0	102	0.0
small_8	23	300	9753	0.001	0.0	9751	0.002
small_9	5	0.0005	130	0.0	0.0	130	0.0
small_10	20	0.008	1025	0.0	0.0	1019.000	0.006
large_1	100	0.032	9147	0.0	0.0	8817	0.036
large_2	200	0.11	11238	0.0	0.0	11227	0.001
large_3	500	3.39	28857	0.0	0.0	28834	0.001
large_4	1000	28.68	54503	0.0	0.0	54386	0.002
large_5	2000	300.001	110547	0.0007	0.0	110547	0.0007
large_6	5000	300.001	256574	0.072	0.001	276379	0.0003
large_7	10000	300.002	248948	0.558	0.004	563605	7.45e-05
large_8	100	0.09	1514	0.0	0.0	1487	0.018
large_9	200	0.456	1634	0.0	0.0	1604	0.018
large_10	500	1.18	4566	0.0	0.0	4552	0.003
large_11	1000	18.46	9052	0.0	0.00051	9046	0.001
large_12	2000	275.22	18051	0.0	0.001446	18038	0.0007
large_13	5000	300.01	44342	0.0003	0.002087	44351	0.0001
large_14	10000	300.004	43935	0.513	0.0025	900200	4.43e-05
large_15	100	0.049	2397	0.0	0.001	2375	0.0092
large_16	200	7.94	2697	0.0	0.0	2649	0.0178
large_17	500	58.30	7117	0.0	0.0	7098	0.0027
large_18	1000	300.007	14390	0.0	0.0	14374	0.0011
large_19	2000	300.007	28919	0.0	0.0015	28827.000	0.0032
large_20	5000	300.01	52541	0.28	0.001	72446	0.00081
large_21	10000	300.003	43419	0.71	0.003	146888	2.11e-04

6.2 Approximation Discussion

The approximation algorithm, which meets the expected time complexity of $O(n \log n)$, ran almost instantaneously for all datasets. The relative error for this algorithm performed better than anticipated; despite a theoretical lower bound error of 50%, the actual error never surpassed 4% in any test except one. The worst performance observed was a 30.4% relative error, suggesting a practical lower bound of 69.6% of the optimal solution, significantly better than expected. Overall, the approximation algorithm excelled in terms of time efficiency and maintained low relative errors throughout the trials.

6.3 Simulated Annealing Discussion

For small instances, Simulated Annealing exhibited exceptional performance with a relative error consistently below 1%. Optimal solutions were found in all 10 runs for most cases. The average runtime of approximately 7 seconds, though seemingly long, is justified by the significant performance improvements, especially in larger instances, provided by the restart mechanism.

In contrast, for larger instances, the relative error remained impressively low, generally below 4%, and even dipped below 1% for the largest datasets. This demonstrates strong performance consistency across different problem sizes, despite the inherent complexity of larger problems. The restart mechanism, though increasing runtime for smaller datasets, proved invaluable for larger instances by allowing the algorithm to effectively escape local minima and explore more of the solution space.

Figures 2 and 5 show how the algorithm quickly finds solutions of relatively good quality for larger datasets. In addition, the graphs show how the algorithm performs better with larger instances.

6.4 Hill Climbing Discussion

The Hill Climbing algorithm demonstrated notable performance, particularly in handling large instances. Figures 8 and 11 showcase

Table 5. Comprehensive Table for the Local Search Algorithms

Dataset	Input Size	Simulated Annealing			Hill Climbing		
		Avg Time (s)	Avg Value	Avg RelErr	Avg Time (s)	Avg Value	Avg RelErr
small_1	10	6.92	295.00	0.00	0.000192	246.80	0.163
small_2	20	7.03	1020.50	0.00	0.000695	938.40	0.084
small_3	4	6.88	35.00	0.00	0.000077	31.10	0.111
small_4	4	6.86	23.00	0.00	0.000078	21.90	0.045
small_5	15	6.95	481.07	0.00	0.000358	399.37	0.170
small_6	10	7.06	52.00	0.00	0.000213	47.40	0.088
small_7	7	6.94	107.00	0.00	0.000147	94.90	0.113
small_8	23	6.94	9743.70	0.00	0.000305	9658.80	0.011
small_9	5	7.02	130.00	0.00	0.000133	122.20	0.060
small_10	20	6.98	1023.30	0.00	0.000686	928.80	0.093
large_1	100	7.16	8811.70	0.04	0.00978	3994.99	0.563
large_2	200	7.14	10896.40	0.03	0.0388	4522.10	0.598
large_3	500	7.23	28080.20	0.03	0.151	7076.75	0.755
large_4	1000	8.08	53812.50	0.01	0.667	10001.70	0.816
large_5	2000	9.08	109763.40	0.01	3.022	17359.60	0.843
large_6	5000	11.13	275776.20	0.00	19.975	32473.90	0.883
large_7	10000	13.34	563056.00	0.00	114.482	58785.20	0.896
large_8	100	7.74	1403.40	0.07	0.00949	1072.50	0.292
large_9	200	7.56	1527.40	0.07	0.0245	1075.10	0.342
large_10	500	8.01	4439.60	0.03	0.103	2638.30	0.422
large_11	1000	8.62	8924.50	0.01	0.346	5236.50	0.422
large_12	2000	9.77	17863.20	0.01	1.473	10403.80	0.424
large_13	5000	11.35	44199.30	0.00	9.414	25454.50	0.426
large_14	10000	13.55	90071.50	0.00	59.232	50440.20	0.441
large_15	100	7.36	2285.00	0.05	0.00903	1363.40	0.431
large_16	200	7.59	2580.90	0.04	0.0225	1364.80	0.494
large_17	500	7.92	6984.70	0.02	0.0804	3236.70	0.545
large_18	1000	8.23	14227.70	0.01	0.235	6199.70	0.569
large_19	2000	8.67	28760.60	0.01	0.956	12079.00	0.582
large_20	5000	10.39	72362.00	0.00	6.262	30025.00	0.586
large_21	10000	13.15	146748.80	0.00	64.195	59569.00	0.595

the algorithm's efficiency at rapidly identifying high-quality solutions for *large_1*, which has an input size of 100. This performance is further highlighted by its scalability and effectiveness, as evidenced in Figures 5 and 6.

However, despite its speed, Hill Climbing's overall efficiency is compromised by its inherent limitations in escaping local maxima. From the analysis presented in Table 5 and the associated graphs, it is evident that while the average completion time for the Hill Climbing algorithm significantly outperforms that of Simulated Annealing, Branch-and-Bound, and approximation algorithms, it does so at a cost to solution quality. The relative error can escalate to 80-90% across a range of random seeds, with the error increasing substantially as the input size grows.

Consequently, for very large input sizes, it is improbable that Hill Climining would achieve an average relative error below 50%, even with a diverse array of random initializations. This highlights a critical vulnerability in the Hill Climbing algorithm: without mechanisms to effectively escape local maxima, its practical utility is limited, particularly when precision is paramount.

6.5 Comparative Analysis

- **Execution Time (sorted by best):** Approximation > Hill Climbing > Simulated Annealing > Branch and Bound
- **Accuracy:** Branch and Bound (small inputs) > Simulated Annealing > Approximation > Hill Climbing
- **Scalability:** Simulated Annealing and Approximation algorithms scale much better with increasing input sizes compared to Branch and Bound and Hill Climbing.

7 CONCLUSION

Overall, there are various algorithms used to approximate the intractable knapsack problem. Empirical evaluation of different knapsack algorithms along with experimentation to improve accuracy have enabled us to determine reasonable quality solutions.

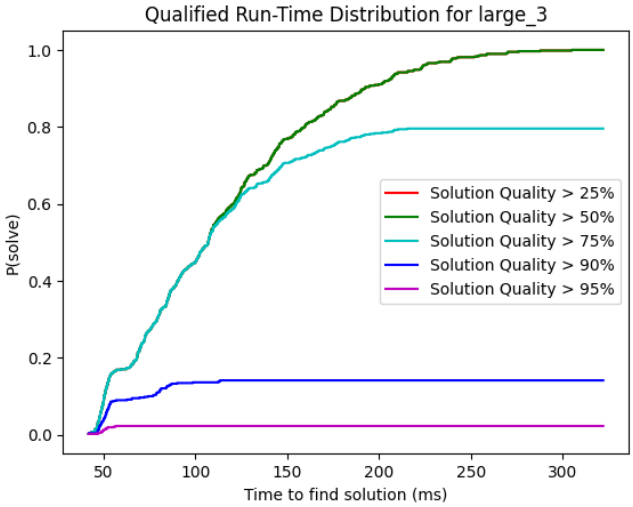

Fig. 11. HC QRTD Graph for large_3 Instance

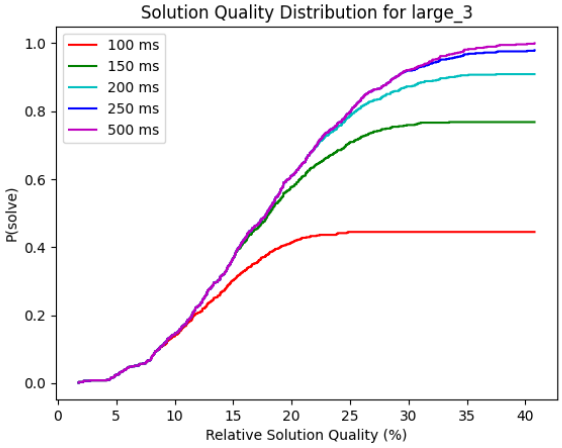

Fig. 12. HC SQD Graph for large_3 Instance

In summary, while each algorithm has its niche—Branch and Bound excelling in small datasets with the possibility of exact solutions, and Hill Climbing in scenarios where time is more critical than accuracy—the Approximation and Simulated Annealing algorithms present the best trade-offs between time efficiency and solution quality across varied input sizes. These insights highlight the importance of choosing the right algorithm based on specific requirements of problem size and desired accuracy.

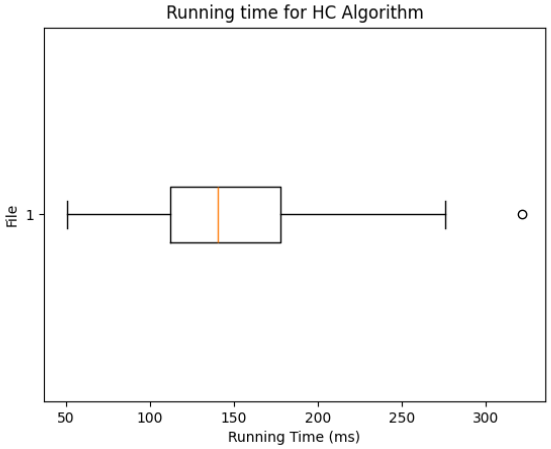

Fig. 13. HC Runtime Boxplot for large_3 Instance

REFERENCES

[1] 2016. 0/1 Knapsack using Branch and Bound. https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/ Section: Branch and Bound.

[2] Samir Balbal. 2015. Local Search Heuristic for Multiple Knapsack Problem. *International Journal of Intelligent Information Systems* 4 (02 2015), 35. https://doi.org/10.11648/j.ijiis.20150402.11

[3] Andrea Bettinelli, Valentina Cacchiani, and Enrico Malaguti. 2017. A Branch-and-Bound Algorithm for the Knapsack Problem with Conflict Graph. *INFORMS Journal on Computing* 29, 3 (Aug. 2017), 457–473. https://doi.org/10.1287/ijoc.2016.0742

[4] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. 2022. Knapsack problems — An overview of recent advances. Part I: Single knapsack problems. *Computers & Operations Research* 143 (July 2022), 105692. https://doi.org/10.1016/j.cor.2021.105692

[5] Chandra Chekuri. 2011. Approximation Algorithms: The Knapsack Problem. Lecture Notes for CS 598CSC at the University of Illinois at Urbana-Champaign. https://courses.engr.illinois.edu/cs598csc/sp2011/Lectures/lecture_6.pdf Lecture date: February 9, 2011, Scribe: Kyle Fox.

[6] Michel Gendreau and Jean-Yves Potvin (Eds.). 2019. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science, Vol. 272. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-91086-4