Assessing Solution Quality and Complexity of Various Algorithms for the 0/1 Knapsack Problem on Real-World Datasets

Keven Yeh[1], Alejandro Danies-Lopez[2], Abhijeet Saraha[3], and Danielle Dowe[4]

[1]School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, Georgia
[2]School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia
[3]School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia
[3]Department of Machine Learning, Georgia Institute of Technology, Atlanta, Georgia

Abstract

This report evaluates four algorithms—Branch-and-Bound, Greedy Approximation, Simulated Annealing (SA), and Hill Climbing (HC)—applied to the NP-complete 0/1 Knapsack problem across a range of problem sizes. This problem, a fundamental challenge in computational theory, is well-known for its complex decision-making under strict constraints. Branch-and-Bound excels in smaller instances by delivering exact solutions but fails in larger scenarios due to exponential state-space scaling. Greedy Approximation offers the fastest solutions, performing optimally in simpler, smaller datasets but showing limitations as complexity increases. SA and HC demonstrate robust performance in larger instances, effectively approximating near-optimal solutions through strategic modifications like random restarts and accepting suboptimal states, thus overcoming the pitfalls of local maxima. The study highlights the importance of selecting an appropriate algorithm based on the problem size and complexity, providing empirical insights that guide real-world applications requiring a balance between computational speed and accuracy. This comparison sheds light on practical trade-offs in computational optimization and enhances understanding of algorithmic efficiency in tackling complex problems.

Keywords: NP-complete, Branch-and-Bound, Approximation, Simulated Annealing, Hill Climbing

1 Introduction

The 0/1 Knapsack Problem remains a significant challenge within the field of combinatorial optimization, attracting considerable attention due to its NP-hard classification and profound implications for real-world applications [1]. This problem necessitates maximizing the value of items within the strict constraints of a knapsack's weight capacity, exemplifying the challenges inherent in resource allocation tasks.

This paper presents a detailed evaluation of four distinct algorithms tailored to address the NP-complete 0-1 Knapsack Problem. Our analysis begins with the 'Branch-and-Bound' method, an exact approach that meticulously partitions solution spaces to ascertain optimal results. We then examine construction heuristics, particularly the 'Greedy Approximation' method, which employs modified greedy techniques to achieve near-optimal solutions with robust approximation guarantees. Additionally, our study assesses local search strategies, including 'Hill Climbing' and 'Simulated Annealing,' which, although lacking guarantees for optimal outcomes, frequently yield results impressively close to the optimum, often surpassing other heuristic techniques. Through rigorous empirical evaluation, we assess the runtime efficiency and solution quality of these algorithms. The aim of our research is to determine which algorithm most effectively balances accuracy with operational speed, thereby enhancing its viability for practical applications where efficient resource allocation is crucial.

2 Problem Statement

The 0-1 knapsack problem is a well-known NP-complete optimization challenge [2]. It involves a knapsack with a specified integer capacity 'C' and a set of n items, which each have a non-negative value v_i and a non-negative weight w_i. The objective is to select a subset of items to maximize the total value without exceeding the capacity of the knapsack. The problem can be written as the following optimization problem:

$$\textbf{max} \sum_{i=1}^{n} v_i x_i \quad \text{subject to:} \quad \sum_{i=1}^{n} w_i x_i \leq C,$$
$$x_i \in \{0,1\} \quad \text{for all } i \in \{1,2,\ldots,n\}$$

In this context, x_i is a binary decision variable where $x_i = 1$ indicates that item i is selected, and $x_i = 0$ indicates it is not. The objective function aims to maximize the total value of the selceted items, while ensuring their combined weight does not exceed C.

3 Related Works

3.1 Exact Algorithm

In reference to the 0/1 Knapsack problem, exact algorithms find the optimal solution. This is done by enumerating all the possible combinations of objects, verifying their validity (total weight $\leq$ max weight) and finding the max value from the possible solutions [3]. This is infeasible to scale up, as the number of possible solutions explodes as the size of the problem increases. Though it is possible to cut down on the number of possible solutions that are checked using properties of substructure of the problem, it is still infeasible for larger size problems.

3.2 Approximation Algorithms

In the context of the 0/1 Knapsack problem, construction heuristics serve as an efficient means to build feasible solutions through systematic procedures. These heuristics are particularly valuable as they often come with theoretical approximation guarantees, providing a measurable level of confidence in the near-optimality of the solutions they generate.

Greedy Heuristic Modification: The conventional dynamic programming (DP) approach to the 0/1 Knapsack problem, while effective in finding an exact solution, operates with pseudo-polynomial complexity and may not be feasible for very large instances or instances where the knapsack's capacity is large relative to the number of items. As an alternative, a modified greedy heuristic can be used to achieve a balance between computational efficiency and solution quality.

This modified greedy heuristic is particularly effective when there are items with disproportionately high values compared to their weights, which is often the case in real-world scenarios where certain items carry significantly more value relative to their cost or size. The simplicity of the approach also allows for rapid execution, making it suitable for applications requiring quick decisions, such as real-time systems or large-scale problems where traditional DP solutions are computationally prohibitive.

3.3 Local search

Local search algorithms are based on heuristics that return relatively "good" solutions, but not necessarily the optimal one [3]. Local search starts with a possible solution in the search space, and iteratively moves from a solution to a "neighboring" solution by making small changes to the initial solution state. This is inspired by exploiting the relative structure of the problem; for example, the potential energy graph of a system. In such a system, it is easy to find the local minimum by making small changes to the state of the system, as the problem graph looks like a "funnel"[3]. Changing the state of the system such that the solution decreases eventually takes the system to a local minimum. In such cases, the algorithm runs until a limit on time or number of steps is reached. This returns a relatively good solution, but there are no guarantees on the optimality of the solution.

4 Algorithms

Four distinct algorithms were implemented to compute the best solution to the 0-1 Knapsack problem: branch-and-bound for an exact solution, Approximation for an approximate solution, and Simulated Annealing and Hill Climbing for local search heuristic solutions.

4.1 Branch-and-Bound

The branch-and-bound algorithm used in this project is what as known as the "least cost" branch-and-bound method of solving the knapsack problem. It shares similarities with the algorithm proposed by Kolesar in his paper on the knapsack problem [4].

Algorithm State-Space: The algorithm works in a state-space of a problem that is formulated as a binary tree. Each node in the tree represents a configuration or a potential solution to the knapsack problem. Here, a configuration is a list of boolean values, where each boolean represents one of the items in the problem. A value of 1 means the item is in the knapsack, and 0 means it is not. The root node of the tree is where all items are in the knapsack. The level of the root node is 0. Additionally, for this algorithm, the order of the boolean values follows the order of the items sorted by descending value-to-weight ratio.

For a problem with N items, there are N levels after the root level. Each parent node has two children, the left child is a copy of the parent, and the right node is the same as the parent, but with one modification: if the right child's tree level is N_c, then the N_c-th item in the parent's knapsack is

removed from the child's knapsack. The first three levels of a tree for a problem with four items is shown in Figure 1.

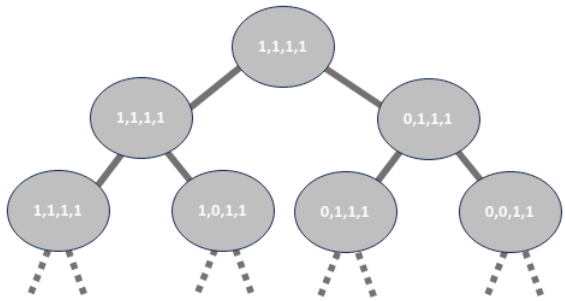

Figure 1: State-Space Tree of a Four-Item Problem

For any given problem, this state-space tree will have $N+1$ levels, which means it will have $2^{N+1}-1$ nodes. However, repeated nodes are always left children, so the algorithm can be constructed in a way that repeated nodes are not re-evaluated. This means that only a maximum of 2^N (the number of distinct configurations in a problem) could ever be evaluated by the algorithm.

Bounding Strategy: Clearly the time complexity of the branch-and-bound algorithm is $O(2^N)$. Any exact solution algorithm to an NP-Hard problem must be exponential. However, the algorithm will ideally not have to explore all 2^N configurations in most problem instances. This can be achieved by not exploring nodes that are known to not contain the optimal solution.

This is done by first multiplying the value of all items by -1, so that the problem is now about minimization instead of maximization. As each node, n, is evaluated, an upper and lower bound for the negative total value of its descendants can be found, denoted as $U(n)$ and $L(n)$ respectively. The smallest upper bound found so far is recorded and denoted UB. If any node has $L(n)$ that is not at least as small as UB, then clearly none of its descendants can contain the optimal solution. The algorithm that finds the upper and lower bounds of a node's state is shown in Algorithm 1.

From Algorithm 1, it can be seen that the upper bound for a node is the value achieved by filling a knapsack in greedy order (highest value-to-weight ratio first) until the capacity is achieved. This will be referred to as the GREEDY-FILL(I, C) method from now on. The lower bound is found the same way, except when the capacity is reached, a fraction of the next best item is added to the knapsack to completely fill the weight capacity. This happens to be the strategy for finding the solution of the fractional knapsack problem. Note that the upper bound is the total value of a state, if the state is valid. Valid meaning that the state's total weight is within the problem weight capacity.

Algorithm 1 Node Bounding Algorithm

```
1:  function BOUND(S, I, C)
2:          ▷ S: n booleans sorted in greedy order
3:          ▷ I: n items of the problem sorted in greedy order
4:          ▷ C: Knapsack weight capacity
5:      L ← 0                                  ▷ Lower Bound
6:      U ← 0                                  ▷ Upper Bound
7:      W ← 0                                  ▷ Total Weight
8:      for i ← 1 to n do
9:          if S[i] is True then
10:             if W + I[i].weight ≤ then
11:                 W ← W + I[i].weight
12:                 L ← L − I[i].value
13:                 U ← U − I[i].value
14:             else
15:                 L ← L − (C − W) I[i].value / I[i].weight
16:                 Break Loop
17:             end if
18:         end if
19:     end for
20:     return L, U
21: end function
```

The Complete Algorithm: The algorithm explores the state-space tree in order of the most promising nodes. The most promising nodes are those with the smallest lower bound. A priority queue is used to accomplish this. If a node's lower bound is greater than UB, then it is not added to the priority queue, which "kills" the node and prevents any of it's descendants from being explored. The algorithm also tracks the best solution found so far. This is the node with the smallest upper bound, out of all nodes explored so far that meet the weight capacity limit. When the state-space has been fully explored, the best solution found so far is guaranteed to be the optimal solution. The full algorithm is shown in Algorithm 2.

Theoretical Complexity: There is no theoretical guarantee that the algorithm will have a smaller time complexity than $O(2^N)$, but in practice it should complete a search more quickly than an algorithm that searches all 2^N possible configurations.

The algorithm does not store the entire state-space tree in a data structure, but the space complexity is still $O(2^N)$ in a worse case scenario. If no nodes are killed, and the algorithm plays out essentially as a breadth-first search, then the maximum number of nodes it can contain are all the nodes in the last level of the tree. This would be $2^{N+1}/2 = 2^N$ nodes. In practice, the algorithm never holds nearly that many nodes.

Algorithm 2 Branch-and-Bound

```
function BRANCH-AND-BOUND(I, C)
                          ▷ I: n items of the problem
                          ▷ C: Knapsack weight capacity
    SORT I by descending value-to-weight ratio
    S ← State/array containing n True values
    L, U ← BOUND(S, I, C)
    UB ← U                ▷ Smallest upper bound seen
    solution ← ∞
    Q ← (L, S, 0) ▷ Priority queue that holds a (lower bound, state, tree level) tuple. Sorts by least lower bound.
    while Q is not empty do
        L, S, level ← Q.POP()          ▷ Parent node
        left ← (L, S, level + 1) ▷ Left child node
        S[level] ← False
        L, U ← BOUND(S, I, C)
        right ← (L, S, level + 1)      ▷ Right child
        if U < UB then
            UB ← U
        end if
        if S is valid and U < solution then
            solution ← U
        end if
        if State is not in a leaf node then
            if left.L ≤ UB then
                Q.PUSH(left)
            end if
            if right.L ≤ UB then
                Q.PUSH(right)
            end if
        end if
    end while
    return solution
end function
```

4.2 Greedy Approximation

The traditional Greedy approach to the 0-1 Knapsack problem, referred to here as 'GreedyKnapsack', arranges items in descending order based on their value-to-weight ratios, such that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$. The algorithm then selects items in this order, ensuring that the addition of each item does not exceed the knapsack's capacity.

However, this method can perform poorly in certain scenarios. For example, consider a case with only two items: the first item with a weight of 1 and a value of 2, and the second item with a weight equal to the knapsack's capacity C and a value also equal to C. GreedyKnapsack would select the first item, ignoring the more valuable second item which alone would provide a better solution.

Thus, in order to address the limitations of the GreedyKnapsack, we introduce the *Modified-Greedy* approach. This enhanced version selects the better outcome between GreedyKnapsack solution and the highest-value individual item. This modification guarantees at least a 2-approximation of the optimal solution.

The algorithm: The Greedy Approximation implemented in this project, complete with approximation guarantees, is detailed in Algorithm 3.

1. *Sort items based on decreasing value-to-weight ratio:* This ratio prioritizes items that offer the highest value for the least amount of weight.

2. *Select items with the highest value per unit of weight:* The algorithm iteratively adds items to the knapsack until adding another item would exceed the knapsack's capacity.

3. *ModifiedGreedy:* A simple modification to the traditional greedy method involves comparing the value of the single most valuable item to the total value of the items initially chosen by the algorithm. Once the knapsack is filled based on optimal value-to-weight ratios, an additional evaluation step is introduced. This step assesses whether the value of any single item exceeds the combined value of the items already selected. If such an item exists, indicating that its value surpasses the combined value of the group, the algorithm adjusts its strategy to select this high-value item as the optimal solution.

Theoretical Analysis: The greedy heuristic inherently provides an approximation guarantee of atleast 50% of the optimal solution. This ratio is derived under the assumption that the optimal solution could potentially fit the two most valuable items partially or fully. Since the greedy algorithm might only select one of these due to capacity constraints, the worst-case scenario would result in the greedy algorithm achieving at least half the value of the optimal solution, where the optimal solution might have fitted both fully or near fully.

Theorem 1. *ModifiedGreedy algorithm achieves an approximation ratio of $\frac{1}{2}$ for the Knapsack problem.*

Proof. We establish this result by first presenting a key claim regarding the performance of the ModifiedGreedy algorithm in comparison to the optimal solution, denoted as *OPT*.

Claim 1. *The partial sum of the values from the first k items, $v_1+v_2+\ldots+v_k$, is at least as great as the optimal solution, denoted OPT. More precisely, it can be shown that $v_1 + v_2 + \ldots + \alpha v_k \geq OPT$, where α is defined as $\alpha = \frac{W-\sum_{i=1}^{k-1} w_i}{w_k}$. Here, α represents the fraction of item k that can still be accommodated in the knapsack after the items 1 through $k-1$ have been packed, where W is the knapsack's*

capacity and w_i denotes the weight of item i. This expression accounts for the remaining capacity in the knapsack and the proportionate contribution of item k to the total value based on available space.

Algorithm 3 Modified Greedy Algorithm

```
function ModifiedGreedy(N, B)
                                   ▷ N: List of items
                ▷ B: Weight capacity of the knapsack

    Sort N by decreasing value-to-weight ratio.

    Initialization of variables
    W ← 0          ▷ Accumulated weight of items
    V ← 0          ▷ Accumulated value of items
    M ← 0          ▷ Maximum single item value

    for each item i in N do

        Verify item fits within capacity
        if W + w[i] ≤ B then
            W ← W + w[i]          ▷ Update total weight
            V ← V + v[i]          ▷ Update total value
        end if

        Verify item is highest value within capacity
        if w[i] ≤ B and v[i] > M then
            M ← v[i]              ▷ Update max item value
        end if
    end for

    Choose greater total value or single max value
    R ← max(V, M)
    return R                      ▷ Return the max possible value
end function
```

The validity of Theorem 1 is apparent from the supporting claim [5]. Specifically, the sum of the values for the first $k-1$ items $(v_1+v_2+\ldots+v_{k-1})$ or the value of the k-th item (v_k) alone must account for at least half of the optimal solution $(\frac{\text{OPT}}{2})$. Therefore, to substantiate Theorem 1, it is essential to establish the credibility of Claim 1. To this end, we introduce a linear programming (LP) relaxation of the Knapsack problem, which is described as follows. Let x_i represent the fraction of item i that is packed in the knapsack, where x_i can take any value from 0 to 1. The objective function to maximize is:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{subject to:} \quad \sum_{i=1}^{n} w_i x_i \leq W$$

$$0 \leq x_i \leq 1 \text{ for all } i \in \{1, \ldots, n\}$$

Let OPT' be the optimal value of the objective function in this LP instance. Since any feasible solution to the original Knapsack problem also meets the criteria for the LP, and both problems share the same objective function, it follows that $\text{OPT}' \geq \text{OPT}$ [5]. By setting $x_1 = x_2 = \ldots = x_{k-1} = 1$ and $x_k = \alpha$, while setting $x_i = 0$ for all $i > k$, we establish a feasible solution to the LP that cannot be further improved by altering any single tight constraint, given that the items are sorted. Consequently, the sum $v_1 + v_2 + \ldots + \alpha v_k$ equals OPT' and is therefore at least as large as OPT. This substantiates the performance guarantee of the approximation algorithm, which assures that it achieves at least 50% of the optimal solution.

The efficiency of the algorithm relies significantly on the initial sorting phase, which is characterized by a time complexity of $O(n \log n)$, where n denotes the number of items. Following this sorting phase, subsequent iterations, which involve randomization and evaluation, operate in linear time with respect to the number of items considered, making the approach feasible for moderately sized datasets. □

4.3 Simulated Annealing

The first of the two local search algorithms that were implemented was a simulated annealing (SA) approach. In a typical SA algorithm for the knapsack problem, the "landscape" of the problem can be though of as a collection of all the different configurations/states that can be found for the input. Each state is surrounded by its neighbors, which are usually the states most similar to it. Every SA algorithm has an evaluation function that assigns a value to each state. The goal is to traverse across the landscape until a global maximum or global minimum is found, using the evaluation function. Since the algorithm is built on heuristics, there are no guarantees that the global optimum will be found. Often times, only a local optimum is obtained.

When deciding which state to move to, a naive approach would be to always move up, in a maximization problem, or always move down, in a minimization problem. However, this can lead to being trapped in a local optimum. The method used to avoid this is where the algorithm gets its name. In physical systems, there is a probability proportional to the Gibbs-Boltzmann function, $e^{-E/(kT)}$, of finding a the system in a state with energy, E [3]. In the function, $T > 0$ is the temperature, and k is a constant. Intuitively, what this means is that as the algorithm progresses (temperature cools) the probability of moving to a state with a worse evaluation metric (higher energy) decreases. This allows the algorithm to escape from local optimums early in

the process.

The Initial State: The SA algorithm can technically start anywhere in the problem landscape. However, it's preferred to start somewhere in the landscape that has a higher likelihood of being closer to the global optimum than a completely random state. For the knapsack problem, one such state is that produced by the greedy fill algorithm, shown in algorithm **??**. This was the initial state chosen for the knapsack SA algorithm used in this project.

Designing the Neighborhood: Every state can be thought of as a collection of boolean values for each item in the problem. A 1 indicates the item is in the knapsack, and a 0 indicates it is not. States that are adjacent neighbors will have items with different boolean values. However, the exact neighbors need to be defined. If it's desirable to have neighbors as similar to each other as possible, then the number of items with different values should be low. Keeping neighbors similar to each other can make it easier to follow the "gradient" of the landscape. However, this also limits mobility in a landscape, because large "jumps" are not possible. Allowing neighbors to have a large number of different boolean values can solve this problem, at the cost of losing the ability to follow gradients closely.

Algorithm 4 Finding a state with m different boolean values

```
1:  function FIND-SIMILAR-STATE(S, m)
2:                  ▷ S: n boolean values of the state
3:      S' ← S
4:      for i ← 1 to m do
5:          k ← random integer between 1 and n
6:          S'[k] ← ¬S'[k]
7:      end for
8:      W ← UPDATE-STATE-WEIGHT(S')
9:      V ← UPDATE-STATE-VALUE(S')
10:     return S', W, V
11: end function
```

Algorithm 4 shows how a similar state is found. For the SA, the value for m is dependent on the size of the problem. Larger problems will have larger landscapes, so m is higher for these problem sizes. The specific values are shown in equation 4. Note that n is the number of items in the problem.

$$m = \begin{cases} 1 & \text{If } n < 3 \\ 2 & \text{If } n < 200 \\ 5 & \text{Otherwise} \end{cases} \quad (1)$$

Finding a state with Algorithm 4 does not guarantee that the new state will be valid. A state is only valid if the number of items in the knapsack don't exceed the knapsack capacity. It is possible to design evaluation functions that penalize states that are invalid, however it is often times better to avoid such states altogether. In the knapsack problem, there can be many more invalid states than valid ones, so avoiding the invalid states can speed up the search process.

One way to ensure a neighbor is a valid state is to use a repair operator on a state found in Algorithm 4. A common repair operator for the knapsack problem is to remove items from the knapsack in a greedy order until the capacity limitation is satisfied [6]. The greedy order, would be to remove items with the smallest value to weight ratio first. However, it's possible to limit the landscape even more by improving a state that is known to be suboptimal. Once a state has been repaired, it can be improved using the same greedy fill algorithm used to find the initial state. This improvement operator simply adds items to the knapsack in order of highest value-to-weight ratio, until the capacity is reached [6]. The pseudo-code for the repair and improvement operators can be seen in Algorithm 5, which also summarizes the entire process for how the project's SA algorithm obtains a neighbor state.

Evaluation Function: Ensuring that the problem landscape only consists of valid states makes the evaluation function for the problem very easy. The function does not need a penalty term to penalize invalid states. The function output can just be the total value of a state. It is clear that with this function the global optimum will coincide with the exact state that maximizes the total value the knapsack can contain.

The algorithm: The entire SA algorithm used for this project can be seen in Algorithm 6. Notice that the algorithm uses a method called "restarting". This is a common heuristic used in SA algorithms to avoid getting trapped in local optimums once the temperature of the problem has cooled down. This method entails restarting the temperature at its initial condition after a certain number of runs have been allowed to pass.

It can also be seen in Algorithm 6 that the constant used to cool the temperature in every iteration is assigned to 0.8 and the number of iterations that are allowed before restarting is 10. The methodology that is employed to find neighbor states allows for significant mobility within the landscape, so allowing the algorithm to run for many iterations before restarting is not necessary. If the initial temperature, T_0, can achieve an 80% average probability to switch to a neighbor state when the neighbor has a lower value, then after 10 iterations this probability will drop to about 12.5%. Note that the initial temperature required to achieve this

will depend on the average value difference between neighbors in the landscape, ΔE_{avg}. Specifically, the ratio $\Delta E_{avg}/T_0$ will need to be about 0.2231.

$$e^{\frac{-\Delta E_{avg}}{T_0}} = e^{-0.2231} \approx 80\% \quad \text{At initial temperature}$$

$$e^{\frac{-\Delta E_{avg}}{t_k^{10} T_0}} = e^{-0.2231\left(\frac{1}{0.8^{10}}\right)} \approx 12.5\% \quad \text{After 10 iterations}$$

Algorithm 5 Find a neighbor state

```
1:  function FIND-NEIGHBOR(S, I, m, C, W, V)
2:                          ▷ S: n boolean values of the state
3:                          ▷ I: n items of the problem
4:                          ▷ m: an integer
5:                          ▷ C: Weight capacity of the knapsack
6:                          ▷ W: Total weight of state S
7:                          ▷ V: Total value of state S
8:
9:      SORT I by decreasing value-to-weight ratio
10:     S', W, V ← FIND-SIMILAR-STATE(S, m)
11:     S', W, V ← REPAIR-STATE(S, m)
12:     S', W, V ← IMPROVE-STATE(S, m)
13:     return S', W, V
14: end function
15:
16: function REPAIR-STATE(S', I, W, V)
17:     for i ← n to 1 in -1 steps do
18:         if W > C and I[i] is in S' then
19:             W ← W − I[i].weight
20:             V ← V − I[i].value
21:             S'[i] ← 0
22:         end if
23:     end for
24:     return S', W, V
25: end function
26:
27: function IMPROVE-STATE(S', I, W, V)
28:     for i ← 1 to n do
29:         if W + I[i].weight ≤ C and I[i] is not in
    S' then
30:             W ← W + I[i].weight
31:             V ← V + I[i].value
32:             S'[i] ← 1
33:         end if
34:     end for
35:     return S', W, V
36: end function
```

However, these probabilities are dependent of finding an appropriate initial temperature for the problem. How is this done? If the Gibbs-Boltzmann function is rearranged, the following can be observed:

$$T_0 = -\Delta E_{avg} / \ln P_0, \tag{2}$$

where P_0 is the initial probability of states switching at ΔE_{avg}. In algorithm 6, the INITIAL-TEMPERATURE() function finds the value difference between neighbors across 100 randomly generated samples of the problem landscape, and then computes the average value. This is an approximation of ΔE_{avg}. The function then computes and returns T_0 using the expression in equation 2.

Algorithm 6 Simulated Annealing for the Knapsack Problem

```
1:  function SIMULATED-ANNEALING(I, C, m, K)
2:                          ▷ I: n items of the problem
3:                          ▷ C: Weight capacity of the knapsack
4:                          ▷ m: an integer
5:                          ▷ K: number of iterations
6:
7:      S, W, V ← GREEDY-FILL(I, C)
8:      O ← V
9:      T ← INITIAL-TEMPERATURE(I, C)
10:     t_k ← 0.8
11:     iter_limit ← 10
12:     c ← 0
13:     for k ← 1 to K do
14:         if c >iter_limit then          ▷ restart occurs
15:             c ← 0
16:             T ← INITIAL-TEMPERATURE(I, C)
17:         end if
18:         S', W', V' ←
19: FIND-NEIGHBOR(S,I,m,C,W,V)
20:         p ← random value ∈ [0,1]
21:         if p < e^((V'−V)/T) then
22:             S, W, V ← S', W', V'
23:             c ← c + 1
24:             T ← T * t_k
25:             O ← MAX(O, V)
26:         end if
27:     end for
28:     return O
29: end function
```

Theoretical Analysis: As with most local search algorithms, the SA algorithm proposed here does not have any guarantees of finding the optimal solution to a problem. However, the initial state used in the algorithm is equivalent to that found in the project's approximation algorithm. So there is a guarantee that the SA algorithm solution will be at least as good as 50% of the optimal solution, but this is not due to any SA heuristics.

The algorithm could potentially have to explore all possible valid knapsack configurations at least once in order to find the optimal solution, which would be $O(2^N)$ where N is the number of items in the problem. However, in practice the algorithm usually finds a near optimal solution in a much shorter amount of time than this.

In terms of space complexity, the proposed SA algorithm is quite efficient. The items in the knapsack problem are stored, but the only states that are

stored are that of the configuration being assessed, one of it's neighboring states, and the state of the best found solution so far. So the space complexity is only $O(N)$.

4.4 Hill Climbing

The second local search algorithm that was implemented used a hill climbing (HC) approach. As before, each subproblem consists of a particular configuration of evaluated items, where some have been selected to be placed into the knapsack. A neighborhood of states that are very similar to the current one is considered, and the algorithm seeks to move in the direction of the neighbor that offers the most improvement in terms of solution quality, determined by a scoring function. The scoring function used for this algorithm is the same as previously described for simulated annealing: the total value of a state. Thus, the algorithm will iteratively move across the solution space in the direction of the closest maximum by changing only a few of the included items at a time (in other words, it will climb the hill it is standing on by moving in the direction the local slope until reaching the summit).

Modifications to the naive HC algorithm: The HC algorithm is, evidently and almost by definition, vulnerable to local maxima. If run on a solution space that has many local maxima, it is very unlikely that it will happen to land on the hill with the global maximum at its summit on its first iteration. In order to account for this shortcoming, three simple modifications were made that improve the robustness of the approach:

1. *Initialize at a random point in the solution space:* Since this algorithm is relatively fast to converge to a local maximum, it doesn't suffer significantly from not being initialized at the result of the greedy solution like the simulated annealing algorithm is. More importantly, this random initialization is necessary because of the following modification:

2. *Random restart when a maximum is reached:* As previously mentioned, a single run of HC can be unlikely to find a global maximum in an unfavorable solution space. After enough iterations, however, the likelihood of reaching the global maximum increases greatly. The algorithm continues restarting at a random position after climbing to the top of a hill until a certain number of restarts is reached without any improvement in the best solution found; this restart limit is set depending on the size of the problem:

$$restart - limit = \begin{cases} 10 & \text{If } n < 200 \\ 40 & \text{If } n < 2000 \\ 100 & \text{Otherwise} \end{cases} \quad (3)$$

3. *Random worse steps:* In order to improve performance within each iteration, there is a certain probability that the algorithm will move to a suboptimal neighbor, where the neighbor's score might be better than that of the current state but worse than the best score in the neighborhood. Unlike in simulated annealing, this probability is fixed for this algorithm and doesn't change as the algorithm progresses. We set this probability at 10% empirically, aiming to improve the robustness of the algorithm without derailing it too far from the path it is on.

Designing the Neighborhood: In order to build a neighborhood for this algorithm, the same Algorithm 6 detailed in the simulated annealing section was used to find each neighbor state. However, unlike SA, which has a chance of moving to each neighbor when it is found, HC requires a neighborhood to be analyzed in its entirety in order to find the neighbor with the highest score within the neighborhood. Obtaining every single neighbor, where the difference between it and the current state is one item that is added, removed, or swapped, can become very expensive as the problem size increases—especially when considering that this must be done for each step that is taken along the gradient. Instead, a neighborhood size limit was set with random neighbors being recruited using Algorithm 6 until reaching the limit, which was determined as follows (n being the number of items in the problem):

$$neighborhood - size = \begin{cases} 50 & \text{If } n < 200 \\ 150 & \text{If } n < 2000 \\ 300 & \text{Otherwise} \end{cases} \quad (4)$$

These thresholds were selected empirically, and allowed us to have a manageable neighborhood size without compromising the algorithm's ability to reach optima.

The algorithm: The HC approach to the knapsack problem was implemented as detailed in algorithm 7:

Algorithm 7 Hill Climbing

```
function HILL-CLIMBING(I, C, m)
                                  ▷ I: n items of the problem
                          ▷ C: Weight capacity of the knapsack
                                          ▷ T: Time cutoff

    Initialize state:
    S ← random value ∈ [0,1]   ▷ State boolean values
    W ← Weight of random selected items   ▷ Total weight of state S
    V ← Value of random selected items   ▷ Total value of state S
    O ← V
    Prev ← V
    r ← 0        ▷ Restarts without improvement
    while time < T and r < restart − limit do
        max − found ← false
        while notmax − found do
            Neighborhood ← []
            for i ← 1 to neighborhood − size do
                S′, W′, V′ ←
FIND-NEIGHBOR(S,I,m,C,W,V)
                Neighborhood ← append S′, W′, V′
            end for
            p ← random value ∈ [0,1]
            if p < 0.1 then
                bestN ← Neighborhood[1]
                if bestN.V < O then
                    S, W, V ←
                    O ← MAX(O, V)
                end if
            else
                bestN ← neighbor with highest V
                if bestN.V < O then
                    S, W, V ←
                    O ← MAX(O, V)
                else
                    max − found ← true
                end if
            end if
        end while
        if O > Prev then
            r ← r + 1
        else
            r ← 0
            Prev ← O
        end if
        Initialize state:
        S ← random value ∈ [0,1]
        W ← Weight of random selected items
        V ← Value of random selected items
    end while
    return O
end function
```

Theoretical Analysis: As a local search algorithm that is vulnerable to unfavorable solution spaces and is initialized randomly, no guarantee can be made about the quality of its solution. The heuristics implemented for the HC algorithm help reduce the probability that it returns solutions that are significantly or extremely suboptimal, and in practice they are successful, but they do not manage to eliminate this probability. Therefore, it is plausible that for some problems, this algorithm will not reliably return a high-quality solution, and that average error is higher than for alternative approaches.

As in simulated annealing, it is possible for this algorithm to have to explore all possible configurations at least one, leading to an $O(2^N)$ time complexity. Fortunately, empirical results are very reasonable, at least with nonzero but very small error. Space complexity is likewise similar to that of SA, with N values needing to be stored for S, the boolean values that indicate a state, giving us an $O(N)$ space complexity. The added necessity to store the neighborhood does not affect this upper bound as implemented, given that a linear number of neighbors is stored with each neighbor taking up $O(N)$; if we did not cap the size of our neighborhood, however, we could have a space complexity of $O(N^2)$ because of the need to store $O(N)$ neighbors.

5 Empirical Evaluation

Experiments involving all four algorithms were run on a collection of datasets with known solutions. These datasets are categorized as small scale (4-23 items), large scale (100-10000 items) and test (5-24 items).

5.1 Experiment Data Collection and Platform

For every run, the best knapsack value found was recorded, along with the items associated with the value's knapsack configuration. Additionally, every time a better solution was found during a run, the solution's value and time to achieve the value were recorded.

Most of the experiments were run on a computer with Ubuntu 22.04, using 32 GB of RAM, and a 13th Gen Intel Processor. The hill climbing experiments were run on a computer with Windows 11 23H2, 8 GB of RAM and an 8th Gen Intel i5 Processor. The algorithms were implemented in Python 3.12.

5.2 Experiment Results

A detailed table of results for all four algorithms is presented in Table 3. The results listed for each dataset are average values derived from 10-20 runs for each algorithm.

To evaluate the solution quality among the studied algorithms, relative error—referred to as 'Rel-Err' in Table 3—was used as the evaluation criterion. Relative error is defined as $\frac{ALG-OPT}{OPT}$, where ALG represents the value of the best solution found for a given instance by a specific algorithm, and OPT is the optimal, maximum value achievable. Consideration is also given to runtime in the evaluation.

Branch-and-Bound: As seen in Table 3, this method demonstrated exceptional accuracy in addressing small problem instances of the 0/1 Knapsack problem, achieving exact solutions with a 0% relative error. This method consistently attained optimum values, highlighting its effectiveness in precise problem-solving under constrained conditions.

Greedy approximation: In our study, we measure the proximity of the greedy algorithm's solution to the optimum via the approximation ratio, a pivotal metric defined as $\frac{\mathrm{OPT(X)}}{\mathrm{A(X)}}$. This ratio quantifies the performance of our algorithm relative to the best possible solution, with a value of 1 representing an exact match. As seen in Table 3, several instances achieved an approximation ratio of 1, indicating a negligible relative error. Specifically, instances such as test_1, small_3, small_5, *small 6*, and small_9 showcase scenarios wherein the greedy algorithm attained optimality, thereby reinforcing the algorithm's efficacy in these contexts.

However, to establish a comprehensive assessment of solution quality, it is crucial to consider the worst-case performance of the algorithm across all instances. The empirical lower bound on solution quality is informed by observing the instance that exhibited the maximum approximation ratio, which was 1.4375 in the case of small_4. This instance represents the most significant deviation from the optimum, thereby setting the lower bound for solution quality. In the context of the 0/1 Knapsack problem, this ratio indicates that the algorithm's solution value is at least $\frac{1}{1.4375} \approx 69.6\%$ of the optimal solution value. serving as the empirical lower bound derived from the algorithm's performance across all evaluated instances.This can be considered the empirical lower bound on the solution quality derived from the algorithm's performance across all tested instances.

To assess how closely the approximation algorithm's solutions approach the true optimum, we analyze the distribution of the approximation ratios as depicted in Figure 2. Notably, the majority of the instances exhibit approximation ratios exceeding 0.95, which suggests that for these cases, the greedy algorithm's solutions are within 5% of the optimal solution. It is in the 'large' category instances that the algorithm demonstrates impressive closeness to the optimal solutions, with approximation ratios consistently near 1. This indicates a strong scaling capability of the greedy approach as the problem size increases, albeit with the caveat that the results could be influenced by the specific characteristics of the items in these larger instances.

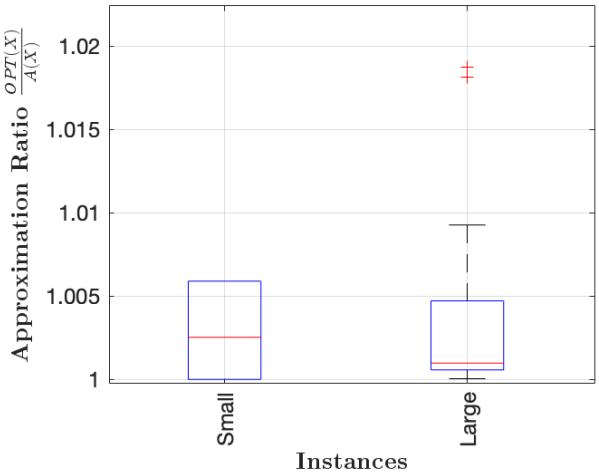

Figure 2: Distribution of Approximation Ratios for Small and Large Instances

Simulated Annealing: For two of the largest data sets, large_20 and large_21, 100 runs of each were performed with the SA algorithm. Plots of the data collected from these runs are shown in Figures 3 through 8.

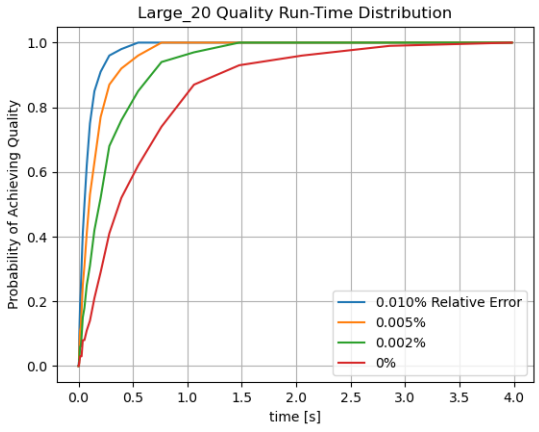

Figure 3: SA Alg. QRTD for a 5000 Item Problem

The quality run-time distributions for large_20

and large_21 are shown in figures 3 and 5 respectively. The time to achieve a 100% probability of finding the optimal solution in large_20 is about 4 seconds, while for large_21 it's about 160 seconds. This suggests that the amount of time to find an optimal solution with this algorithm scales at an exponential rate, because large_21 contains only twice as many items as large_20. The time to achieve a specific relative error follows the same pattern as well.

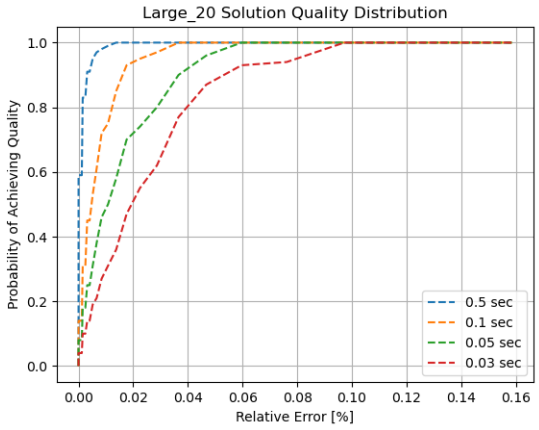

Figure 4: SA Alg. SQD for a 5000 Item Problem

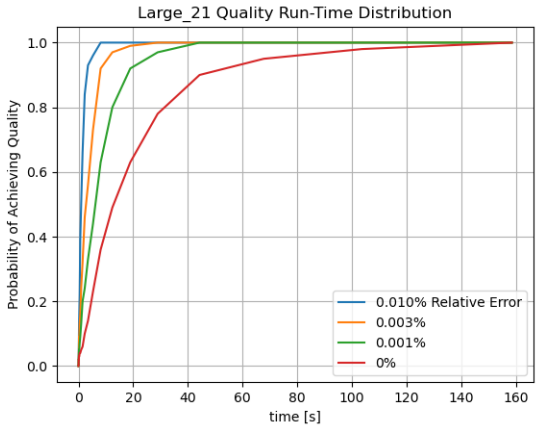

Figure 5: SA Alg. QRTD for a 10000 Item Problem

The relative errors that are shown in figures 3 and 5 are quite small, with the largest being 0.010%. This suggests that the SA algorithm can find a near-optimal solution very quickly in comparison to the time it takes to find the actual optimal solution for a problem. For example, the algorithm has a near certainty to find a solution within 0.010% error in 0.5 seconds for large_20 and about 10 seconds for large_21. This is likely due to the greedy heuristic used to find an initial state in the algorithm. Before the algorithm even begins iterating, the initial state will often be at a near-optimal value.

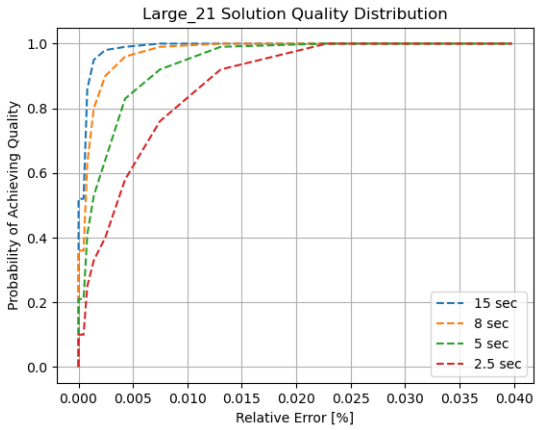

Figure 6: SA Alg. SQD for a 10000 Item Problem

The boxplots for large_20 and large_21 in figures 7 and 8 show data about the run-times taken to achieve the exact optimal solution, using SA. For both datasets, the distribution of run times is skewed significantly, and have outliers with an extreme amount of variation. In both datasets, the second quartile has nearly half the amount of variation as the third quartile. Additionally, all outliers are in the fourth quartile, which means that there are no outliers where the optimal solution was found early. The median time for both data sets is also less than the average, shown in table 3.

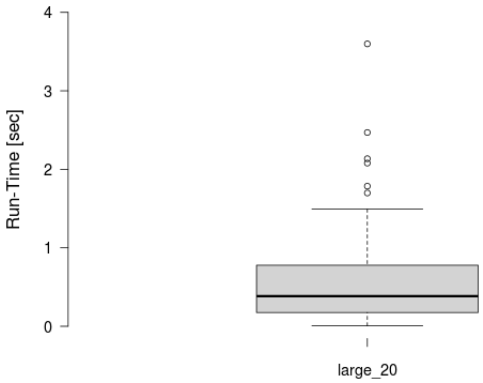

Figure 7: SA Run-times of large_20

This shows that in most cases the SA algorithm finds an optimal solution relatively quickly, but in the cases where it is not found quickly, it often takes a very long amount of time (up to 5 times the average) to find it. There is a degree of inconsistency with the performance of the algorithm on larger datasets.

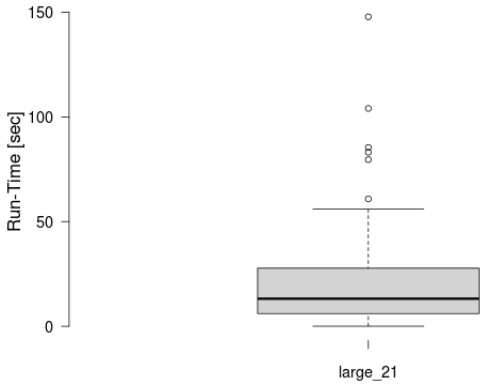

Figure 8: SA Run-times of large_21

The boxplot statistics for the SA algorithm shown in Table 1 reiterate this point. In both data sets, the lower whisker and median are very close values, while the upper whisker is significantly larger.

	large_20	large_21
Upper whisker	1.49	55.95
3rd quartile	0.78	27.74
Median	0.38	13.16
1st quartile	0.17	6.06
Lower whisker	0.00	0.03
Nr. of data points	100.00	100.00

Table 1: Boxplot Statistics for Simulated Annealing

Hill Climbing: 10 runs of the same problems, large_20 and large_21, were also performed with the HC algorithm, as shown in Figures 9 through 14.

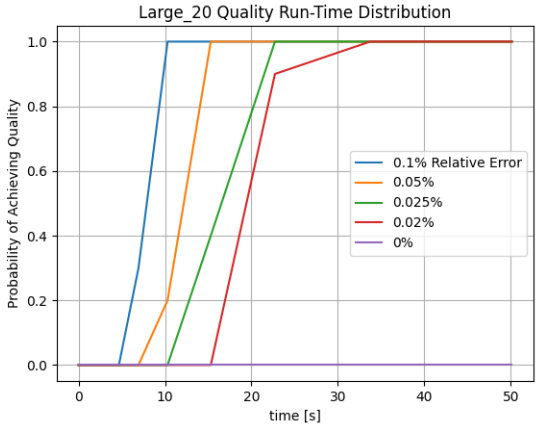

Figure 9: HC Alg. QRTD for a 5000 Item Problem

Unlike the simulated annealing algorithm, the HC algorithm did not manage to find an optimal solution for either of these problems, making them useful illustrative examples of the differences between these two algorithms. Notably, there is a threshold beyond which no lower error was found, being 0.0108% for problem large_20 and 0.0122% for problem large_21. Above these thresholds, more and more solutions are found more quickly as the error bound relaxes, as can be seen in figures 9 and 11. These error thresholds also took longer to converge to near 1 probability than those achieved by SA, suggesting that SA may be a better choice to design an algorithm for this problem.

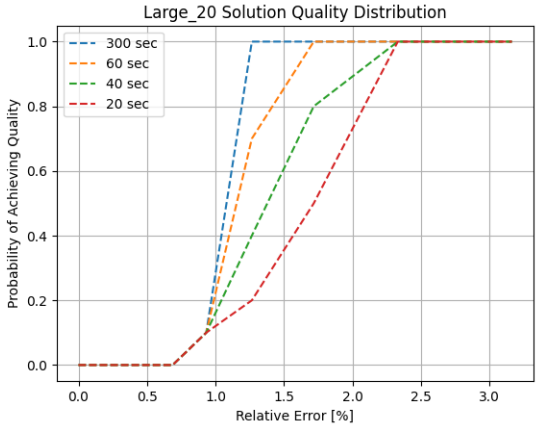

Figure 10: HC Alg. SQD for a 5000 Item Problem

It can also be observed in the SQD plots in Figures 10 and 12 that even running the HC algorithm for five and ten minutes, respectively, was not enough to achieve an optimal solution. This is likely attributable to the particular shape of these problems; it is likely that there are local maxima close to the global maximum that pull the algorithm away from it. It is possible to adjust the algorithm in order to avoid these pitfalls, but that would entail running the risk of overfitting to this type of problem, and worsening performance for most other problems.

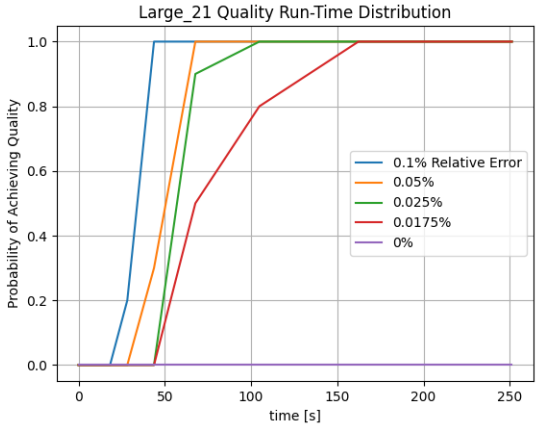

Figure 11: HC Alg. QRTD for a 10000 Item Problem

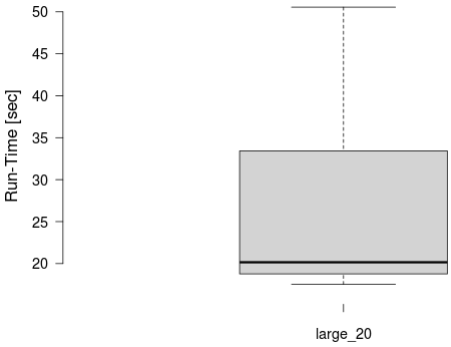

Figure 13: HC Run-times of large_20

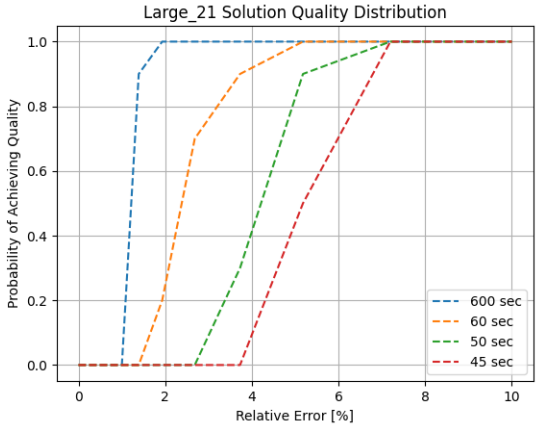

Figure 12: HC Alg. SQD for a 10000 Item Problem

Both box plots are skewed towards their lower ends, with the median and first quartile being very close to each other. There are some notable outliers with much larger running times in both, which is coherent with the stochastic nature of the algorithm. large_21 did have a lower whisker that is considerably faster than even its first quartile, demonstrating that random initialization can occasionally result in a very good result much sooner than would otherwise be expected.

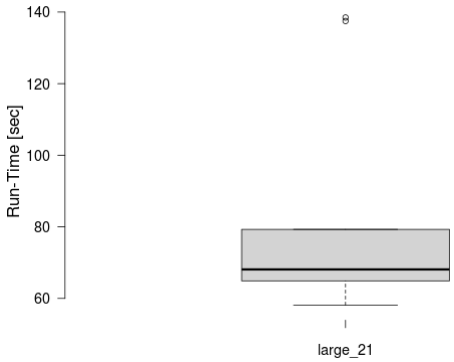

Figure 14: HC Run-times of large_21

It is important to note that, since the HC algorithm did not manage to find an optimal solution for these two problems, the following box plots in figures 13 and 14 were instead found with the time to find a solution with 1.75% error for both problems. This threshold was chosen in order to allow for a wider variety of running times, whereas a tighter threshold would have risked having many iterations be unable to reach it.

Another notable result is that the upper whisker and third quartile for large_21 have the same value. This highlights how far the outlier around 140 seconds is from the rest of the samples, which were all much more tightly clustered.

	large_20	large_21
Upper whisker	50.56	79.29
3rd quartile	33.43	79.29
Median	20.16	68.10
1st quartile	18.77	64.90
Lower whisker	17.51	58.12
Nr. of data points	10.00	10.00

Table 2: Boxplot Statistics for Hill Climbing

6 Discussion

6.1 Branch-and-Bound For Large Problems

As seen in Table 3, the branch-and-bound algorithm always finds the exact solution to the problems with sizes < 25. However, no solution at all was found for the larger problems. Due to the exponential scaling of the problem, it's clear that finding the optimal solution is not feasible with branch-and-bound. But the algorithm was designed to deliver the best found solution, so if it's given a cutoff time, then one would think there would still be some sub-optimal solution returned.

During the experiment, branch-and-bound was run on large_1, a 100 item problem, for 8 hours, and not even a sub-optimal solution was found. The explanation for this is discussed in the following paragraphs.

States Explored By Branch-and-Bound: Recall, that for every knapsack problem there are 2^N possible ways to fill the knapsack while ignoring the weight capacity. So there are a maximum of 2^N states that must be explored to find the optimal solution, in a naive exact algorithm. Branch-and-bound allows for some of these states to be eliminated, so only a percentage, P, of the 2^N states must be explored.

Within the 2^N possible states of a problem, only a portion of them will be valid solutions that meet the weight capacity requirement. These are the solutions that the branch-and-bound algorithm could return. An upper bound can be found on the number of these solutions for a given problem. If the items are sorted in order of increasing weight, then a valid state can be found by adding the items to the knapsack in this order until the weight capacity is reached. This state contains the maximum number of items that any state in the state-space could hold. Denote this number as I_{max}. An upper bound on the number of valid solutions in the state space, S_{valid} would be the number of states that contain I_{max} or fewer items. This is shown in equation 5.

$$S_{valid} \leq \sum_{i=1}^{I_{max}} \binom{N}{i} \tag{5}$$

In the number of states explored by branch-and-bound, $P2^N$, there is a maximum of S_{valid} of these states being valid solutions. So if the computational time to process a node, T_{node} is known, then the average time to find a valid solution, T_{avg}, is lower bounded by

$$\begin{aligned} T_{avg} &\geq \left(\frac{P2^N}{S_{valid}}\right) T_{node} \\ &\geq \left(\frac{P2^N}{\sum_{i=1}^{I_{max}} \binom{N}{i}}\right) T_{node}. \end{aligned} \tag{6}$$

Time to Find a Solution in large_1: The problem in large_1 and previous experimental data can be used to find the minimum average time that it would take for branch-and-bound to find any solution to large_1.

In the experiment runs on small_8, of its 2^{23} distinct states, 5304908 of them had to be evaluated using branch-and-bound. As seen in table 3, it took the algorithm about 23.40 seconds on average to complete this evaluation, which means that each node takes an average of $\frac{23.40}{5304908} \approx 4.411 \times 10^{-6}$ seconds to evaluate on the hardware platform used.

For large_1, the values of N and I_{max} are 100 and 13, respectively. A value of 0.001 can arbitrarily be assigned to P, which means that only 0.1% of the states in large_1 would need to be evaluated by the algorithm. This is not likely the actual value of P for the problem, but it is a generous assumption to make. This means that whatever average time is calculated with this number, the actual is likely much larger. For example, the value of P for small_8 is $\frac{5304908}{2^{23}} \approx 63.2\%$.

So then equation 6 can be used to bound T_{avg} for large_1:

$$\begin{aligned} T_{avg} &\geq \left(\frac{0.001(2^{100})}{\sum_{i=1}^{13} \binom{100}{i}}\right) 4.411 \times 10^{-6} \left(\frac{1 \text{ hr}}{3600 \text{ sec}}\right) \\ &\approx 1282.1 \text{ hr} \end{aligned}$$

So even if the branch-and-bound algorithm can eliminate 99.9% of all states that need to be explored in large_1, the average amount of time to find *any* solution (even a sub-optimal one) for large_1 is clearly infeasible on the hardware used in this project.

Expected Time Complexity: The runtimes for the datasets presented in Table 3 do not appear to necessarily increase with problem size. While this pattern does not indicate a time complexity of $O(2^N)$, the inability to find solutions for larger problem sizes does imply some form of exponential growth.

The reason the $O(2^N)$ complexity is not observed here is that the branch-and-bound heuristic does not consistently "shrink" the state-space of every problem instance. Some problem state-spaces are shrunk more than others. For instance, the branch-and-bound algorithm had to evaluate $5304908/2^{23}$ distinct states for small_8, but only had to evaluate $44/2^{24}$ distinct states for small_10. This explains why the average time to complete small_8 is significantly longer than that of small_10, despite small_10 having twice as many distinct states. This massive variation in the number of states that must be evaluated is problem dependent and difficult to predict.

6.2 Greedy Approximation

The greedy approximation algorithm consistently demonstrates the shortest runtime among the compared algorithms, as evidenced by the data provided in Table 3. This efficiency arises from its algorithmic nature, which relies on making sequential decisions based solely on immediate, locally optimal choices without revisiting or revising past decisions. This characteristic makes it particularly effective for smaller instances where the solution space is not exceedingly complex.

While the greedy algorithm is the fastest, it does not guarantee an optimal solution, particularly in complex or large instances where the sequence of choices heavily influences the overall outcome. However, in smaller instances like as test_1, small_3, small_5, small_6, and small_9, which demonstrated a relative error of 0% in Table 3, the greedy algorithm achieved optimal solutions, precisely matching the best possible outcomes. This level of accuracy highlights the algorithm's robustness in scenarios where the decision-making process can effectively capitalize on simpler problem structures without compromising solution quality.

Expected Time Complexity: As previously discussed, the theoretical time complexity of the Greedy Approximation algorithm is expected to be $O(n \log n)$, primarily due to the sorting step that precedes item selection. The data presented in Table 3 supports this expected complexity. Despite an increase in the number of items, and presumably in the complexity of decision-making from small to large instances, the increase in runtime does not appear to be linear or exponential relative to the number of items. Instead, the modest increase in runtime, even for significantly larger problems, suggests a complexity closer to $O(n \log n)$, where the logarithmic factor contributes to efficient management of larger datasets.

6.3 Local Search and Algorithm Comparison

As shown in Table 3, the SA and HC algorithms achieved the optimal solution for problems with sizes < 25 in approximately the same amount of time as the branch-and-bound algorithm. However, since the branch-and-bound algorithm guarantees finding the optimal solution—a feature not shared by local search algorithms—it is not in these instances where they excel. Instead, their expertise becomes apparent in larger problems (sizes 100-10000), where they are able to achieve near-optimal solutions (and often times *the* optimal solution).

For larger problems, the relative errors for both SA and HC are consistently well under 1%. However, the time required to achieve these results can vary significantly. Some problems can be resolved in less than a second (with fewer than 1000 items), while others may take several minutes. For problems of this scale, they clearly outperform other algorithms, despite lacking strict theoretical guarantees.

Their relative errors are usually an order of magnitude lower than those of the Approximation algorithm on the same dataset. Moreover, the branch-and-bound algorithm cannot feasibly be run on the larger datasets within a reasonable timeframe. Given that the branch-and-bound algorithm remains quick for smaller problem sizes, there is no compelling reason to use any of the other algorithms for problems that are not large.

The Approximation algorithm does not appear to offer any performance advantages that are not already achieved by the local search algorithms. It finishes very quickly, even for large problem sizes, but the QRTD and SQD plots for SA and HC indicate that they can attain a solution quality that surpasses that of the Approximation algorithm within similar timeframes. Moreover, in comparisons between SA and HC, SA slightly outperforms in almost every dataset, both in terms of runtime and quality. This was anticipated, as simulated annealing algorithms are typically more effective at escaping local maxima than hill-climbing mechanisms.

Table 3: Comprehensive Analysis of Performance Across Four Distinct Algorithms.

Instances	Branch and Bound			Greedy Approximation			Simulated Annealing			Hill Climbing		
	Time	Best Sol	RelErr	Time	Best Sol	RelErr	Time	Best Sol	RelErr	Time	Best Sol	RelErr
test 1	2.85e-4	309	0	3.10e-6	309	0	3.86e-6	309	0	2.80e-4	309	0
test 2	5.50e-5	51	0	2.15e-6	47	7.84	2.04e-5	51	0	2.90e-4	51	0
test 3	2.32e-4	150	0	1.67e-6	146	2.67	5.86e-5	150	0	2.96e-4	150	0
test 4	1.50e-4	107	0	3.10e-6	102	4.67	8.49e-5	107	0	4.02e-4	107	0
test 5	1.97e-4	900	0	5.01e-6	858	4.67	1.20e-4	900	0	4.37e-4	900	0
test 6	3.91e-4	1735	0	2.15e-6	1478	14.80	1.42e-4	1735	0	4.43e-4	1735	0
test 7	2.68e-3	1458	0	4.77e-6	1441	1.17	2.66e-4	1458	0	1.32e-3	1458	0
test 8	0.14	13549094	0	6.19e-6	13415886	0.98	0.13	13549094	0	5.13e-2	13549094	0
small 1	5.95e-4	295	0	5.01e-6	294	0.34	1.74e-4	295	0	3.61e-4	295	0
small 2	5.54e-4	1024	0	6.91e-6	1018	0.59	4.12e-4	1024	0	1.04e-3	1024	0
small 3	4.47e-5	35	0	1.91e-6	35	0	3.14e-6	35	0	1.09e-4	35	0
small 4	5.56e-5	23	0	1.91e-6	16	30.43	6.84e-5	23	0	2.59e-4	23	0
small 5	5.49e-4	481.0694	0	5.01e-6	481.0694	0	4.01e-6	481.0694	0	4.83e-1	481.0694	0
small 6	2.60e-4	52	0	3.10e-6	52	0	3.90e-6	52	0	2.11e-4	52	0
small 7	2.71e-4	107	0	1.91e-6	102	4.67	9.61e-5	107	0	3.60e-4	107	0
small 8	23.40	9767	0	5.72e-6	9751	0.16	3.10e-4	9767	0	1.18e-3	9767	0
small 9	4.80e-5	130	0	1.91e-6	130	0	3.19e-6	130	0	1.81e-4	130	0
small 10	4.47e-4	1025	0	7.15e-6	1019	0.59	3.84e-4	1025	0	1.19e-3	1025	0
large 1	-	-	-	5.96e-6	8817	3.61	7.74e-2	9147	0	3.87e-3	9147	0
large 2	-	-	-	6.91e-6	11227	0.10	2.61e-2	11238	0	8.79e-2	11238	0
large 3	-	-	-	9.78e-6	28834	0.08	48.58	28857	0	25.29	28857	0
large 4	-	-	-	1.79e-5	54386	0.21	0.22	54503	0	1.72	54503	0
large 5	-	-	-	2.72e-5	110547	0.07	5.84	110619	5.42e-5	13.43	110619	5.42e-5
large 6	-	-	-	1.35e-4	276379	0.03	32.76	276452.9	1.48e-5	160.32	276442.3	5.31e-5
large 7	-	-	-	1.25e-4	563605	7.45e-3	21.72	563641.1	1.05e-5	179.50	563519.5	2.26e-4
large 8	-	-	-	7.87e-6	1487	1.78	1.92e-3	1514	0	2.23e-1	1514	0
large 9	-	-	-	1.22e-5	1604	1.84	0.3892	1634	0	5.46e-1	1633.75	1.53e-4
large 10	-	-	-	2.10e-5	4552	0.09	2.44e-2	4556	0	3.46	4566	0
large 11	-	-	-	3.10e-5	9046	0.07	344.11	9052	0	31.46	9052	0
large 12	-	-	-	7.72e-5	18038	0.07	12.54	18050.3	3.88e-5	45.16	18050	5.54e-5
large 13	-	-	-	2.62e-4	44351	0.01	24.97	44354.7	2.93e-5	100.75	44353.4	5.86e-5
large 14	-	-	-	8.06e-4	90200	4.43e-3	25.51	90202	2.22e-5	130.78	90175.6	3.15e-4
large 15	-	-	-	6.19e-6	2375	0.92	0.24	2397	0	4.05e-3	2397	0
large 16	-	-	-	6.91e-6	2649	1.78	1.90e-2	2697	0	5.22e-2	2697	0
large 17	-	-	-	9.78e-6	7098	0.27	0.15	7117	0	4.05	7117	0
large 18	-	-	-	1.82e-5	14374	0.11	0.57	14390	0	17.76	14390	6.95e-3
large 19	-	-	-	3.10e-5	28827	0.32	0.11	28919	0	59.73	28758.8	9.99e-1
large 20	-	-	-	6.70e-5	72446	0.08	0.57	72505	0	153.80	71723.3	1.08e-2
large 21	-	-	-	1.27e-4	146888	0.02	20.55	146919	0	227.02	145126	1.22e-2

Note: Time is represented in seconds; RelErr represents the relative error percentage; Best Sol refers to the best solution quality attained.

7 Conclusion

This study's evaluation of four algorithms addressing the NP-complete 0-1 Knapsack problem has significant implications for computational optimization. Despite the complexity of NP-complete problems, these algorithms offer viable solutions that balance implementation ease, runtime efficiency, and solution accuracy. Real-world applications often permit slight deviations from the optimal solution in exchange for more reasonable runtimes. For instance, while Branch-and-Bound provides exact solutions and is unbeatable for smaller datasets, its lack of scalability makes it less suitable for larger problems. In contrast, heuristic approaches like Simulated Annealing and Hill Climbing efficiently handle larger datasets by delivering high-quality solutions that closely approximate the optimum, making them ideal for complex scenarios where absolute precision is less critical. The Greedy Approximation excels in scenarios demanding rapid decision-making, offering a practical compromise between speed and accuracy. This exploration of trade-offs underlines the practical applications of these algorithms in real-world settings, suggesting that depending on the specific requirements and constraints, different algorithms may be preferable. Future research could explore enhancements such as parallel processing to further improve the efficiency and applicability of these methods.

References

[1] M. Monaci, C. Pike-Burke, and A. Santini, "Exact algorithms for the 0–1 time-bomb knapsack problem," *Computers Operations Research*, vol. 145, p. 105 848, 2022.

[2] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Berlin: Springer, 2004, p. 57.

[3] J. Kleinberg and E. Tardos, *Algorithm Design*. Pearson Education, 2006.

[4] P. J. Kolesar, "A branch and bound algorithm for the knapsack problem," *Management Science*, vol. 13, no. 9, 1967.

[5] E. L. Lawler, "Fast approximation algorithms for knapsack problems," *Mathematics of Operations Research*, vol. 4, no. 4, pp. 339–356, 1979.

[6] N. Moradi, V. Kayvanfar, and M. Rafiee, "An efficient population-based simulated annealing algorithm for 0-1 knapsack problem," *Engineering with Computers*, 2022. [Online]. Available: https://doi.org/10.1007/s00366-020-01240-3.