CSE 6140 Final Project Report

Aarosh Dahal
School of Civil and Environmental Engineering
Georgia Institute of Technology
adahal8@gatech.edu

Praful Patil
School of Civil and Environmental Engineering
Georgia Institute of Technology
ppatil72@gatech.edu

Joshua Wood Reeves
School of Computer Science
Georgia Institute of Technology
jreeves49@gatech.edu

Adam Siffel
School of Mechanical Engineering
Georgia Institute of Technology
asiffel3@gatech.edu

1 INTRODUCTION

The Knapsack Problem is a common NP-Complete problem in combinatorial optimization. The goal is to pick a subset of items, each with its own value and weight, to maximize the total value without exceeding the weight limit. We have applied four different algorithms to this problem: Branch and Bound, Approximation, Hill Climbing Local Search, and Simulated Annealing Local Search. Branch and Bound is similar to brute force except it uses a tree-like structure where nodes can be pruned if they are not promising. Unlike Branch and Bound, Approximation will not always find the optimal solution, but it is generally faster and can guarantee an approximation ratio bound. Local Search algorithms iteratively move from a current solution to neighboring solutions using evaluation functions. While they cannot guarantee an approximation ratio, they are generally effective in practice. Our results from these four algorithms point towards the Approximation algorithm being the most effective algorithm due to its speed and low relative error. Branch and Bound is too computationally expensive while the Local Search algorithms do not perform well on large datasets.

2 PROBLEM DEFINITION

For the Knapsack Problem, given a weight limit W and n items where each item i has a value $v_i > 0$ and weight $w_i > 0$, find the subset of items that maximizes the total value without going over the weight limit W. Formally, the optimization objective is: $\max_S \sum_{i \in S} v_i$ subject to $\sum_{i \in S} w_i \leq W$ where S is a subset of items. The decision problem is to determine if there is a subset of items where $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq V$ where V is a target value.

3 RELATED WORK

One Dimensional Knapsack Problem (KP01) has been a well known problem in the field of combinatorial optimization. Except for a few particular cases/ variants of the Knapsack Problem [1], the problem is, in general, NP Hard. Among many algorithms that attempt to solve KP01, Branch and Bound is one of the exact algorithms. [2] proposed a branch and bound algorithm which sorts items greedily, and iteratively selects and removes items to maximize profit within weight constraints, and continually updates the best solution until exhaustive search yields no further improvement. [3], on the other hand used a different upper bound, while [4] employed a two-phase forward move for item selection to efficiently update the current solution while avoiding unnecessary backtracking, and utilizes a parametric technique for computing upper bounds by storing information related to the current solution. To reduce the storage requirement of such algorithms, [5] have developed an algorithm that splits the problem into sub problems (similar to dynamic programming). This method was found to work well for less number of items with large weights and values.

Due to the large space and time complexity of exact methods, near optimal methods such as greedy or greedy-like heuristic methods have been devised [6], which significantly improve computational efficiency of the existing methods and generates robust and near-optimal solutions. Similarly, local search approaches have been developed in attempts to solve KP01 which try to minimise an objective function such as Population Based Simulated Annealing [7] , tabu search (TS) based on strategic oscillation, and surrogate constraint information [8].

4 ALGORITHMS

4.1 Branch and Bound

Description. The Branch and Bound (BnB) algorithm uses a tree-like search of candidate solutions where subtrees can be pruned if they cannot potentially lead to better results than those already found. Given enough time, BnB will always find the optimal solution and is almost always faster than Brute Force, although it is typically slower than other algorithms such as Approximation and Local Searches.

For our BnB, we first sort the items in decreasing order of value-to-weight ratio to save on computation in the lower bound function. Then, all values are converted to negative in order to make it a minimization problem. If the number of items are over 50, BnB uses the approximation algorithm to get a starting upper bound on the solution due to the computational load of large datasets.

The main algorithm for BnB can be seen in Algorithm 1. The upper bound (UB) is simply computed as the max total value that has been seen so far to fit in the Knapsack. The lower bound (LB), as seen in Algorithm 2, uses the greedy algorithm for the Fractional Knapsack Problem to compute the lower bound for a current node/solution. The greedy algorithm picks objects in descending order of value-to-weight ratio and can take a fraction of an object to fill the rest of the knapsack. A priority queue is used to order the nodes from most promising to least promising using the lower bound for comparison.

Time and Space Complexity. The time complexity for BnB is $O(2^n)$ because it uses a binary tree with a maximum depth of n. Each node has two children, which correspond to using the next item or not, and the algorithm could explore every possible node where the maximum number of nodes in a binary tree is $O(2^n)$.

The space complexity is also $O(2^n)$ because the entire tree could be explored and each node would need to be saved.

4.2 Approximation

Description. The approximation algorithm implemented is a modified greedy algorithm that normally produces optimal solutions for the Fractional Knapsack Problem, but not the 0-1 Knapsack Problem we are solving.

Algorithm 1: Branch and Bound Main

```
Data: A list of items with length N and weight
      limit W
Result: A list of selected items
$sorted\_items \leftarrow sort(items)$ in decreasing order,
  then convert to negative values;
$UB \leftarrow 0$ if $N < 50$ else $approximation()$;
$PQ \leftarrow$ PriorityQueue with empty root node;
while $PQ$ not empty do
    $curr\_node \leftarrow$ best node from PQ;
    if $LB(curr\_node) < UB$ then
        if Next depth $< N$ then
            $right \leftarrow$ node for ignoring next item;
            Add $right$ to PQ;
            if $total\_weight + item_i \leq W$ then
                $left \leftarrow$ node for adding next
                  item to knapsack;
                if $total\_value + left < UB$ then
                    $UB \leftarrow total\_value + left$;
                end
                Add $left$ to PQ;
            end
        end
    end
end
return UB;
```

Algorithm 2: Branch and Bound Lower Bound

```
Data: Takes a node with total value and weight
      and sorted items
$i \leftarrow node.depth + 1$;
while $i < N$ AND $total\_weight \leq W$ do
    Add $item_i$ value to $total\_value$;
    Add $item_i$ weight to $total\_weight$;
    $i \leftarrow i + 1$
end
if $i < N$ then
    $total\_value \leftarrow total\_value + \frac{item_i.value}{item_i.weight} *$
      $(W - total\_weight)$;
end
return $total\_value$;
```

Two greedy algorithms are run separately: one that sorts the items by their density (value to weight ratio) and one that sorts the items by their value. Each one then greedily takes the next item that fits into the knapsack.

The total value found by each greedy algorithm is then compared and the higher one is taken.

Each greedy approach naively seems like a good approach, but both can perform very poorly given the right (or rather wrong) set of items. Running both and taking the higher of the two ensures a better result. Doing so also ensures a guarantee ratio to the optimal solution value.

Approximation Guarantee. This algorithm has a guarantee ratio of 2, meaning the solution will be at least 1/2 of the optimal.

Proof:

Suppose greedy algorithm takes items $1, 2, ..., k-1$ but not k^{th} item from sequence sorted by density. Then $\sum_{i=1}^{k} v_i > OPT(ILP)$.

$$\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \leq W \forall 0 \leq x_i \leq 1$$

From LP relaxation: $OPT = OPT(ILP) \leq OPT(LP)$

$$\sum_{i=1}^{k} v_i > OPT(LP)$$

$$OPT(ILP) \leq OPT(LP) < \sum_{i=1}^{k} v_i$$

$$\sum_{i=1}^{k} v_i = \sum_{i=1}^{k-1} v_i + v_k > OPT$$

At least one of greedy approaches is $> \frac{OPT}{2}$.

Time and Space Complexity. The time complexity of this algorithm is $O(n \log n)$, with the dominating component being the sorting step. The space complexity is $O(1)$ as no additional storage is required based on n.

Algorithm 3: Approximation

```
Data: $items, capacity$
Result: $totalValue, selectedItems$
Greedy approach 1
$totalValue1, totalWeight1 \leftarrow 0$
 $selectedItems1 \leftarrow []$
$items$.sort by density in reverse
for $eachitem$ do
    if $totalWeight1 + item.weightis \leq capacity$
     then
        $totalValue1 += item.value$
         $totalWeight1 += item.weight$
         $selectedItems1.append(item.index)$
    end
end
Greedy approach 2
$totalValue2, totalWeight2 \leftarrow 0$
 $selectedItems2 \leftarrow []$
$items$.sort by value in reverse
for $eachitem$ do
    if $totalWeight2 + item.weightis \leq capacity$
     then
        $totalValue2 += item.value$
         $totalWeight2 += item.weight$
         $selectedItems2.append(item.index)$
    end
end
Output better result
if $totalWeight1 > totalWeight2$ then
    return $totalValue1, selectedItems1$
else
    return $totalValue2, selectedItems2$
end
```

4.3 Local Search 1

Description. For the first local search approach, hill climbing algorithm is implemented. In initial attempts, best improvement search was implemented to find the best combination of switching the selection of two items in the knapsack. However, due to high time complexity of this algorithm ($O(n^3)$), random iterative improvement was preferred. This algorithm randomly selected two items and inverts their selection in the knapsack; the neighbor is accepted if the value exceeds that of current solution. In the first step, since solution is initialised randomly, the solution is penalised by returning negative value for weight exceeding capacity.

Time and Space Complexity. When the algorithm runtime is governed by number of iterations
Time Complexity = $O(n^2)$
Space Complexity = $O(n)$, class *Item* size

4.4 Local Search 2

Description.

Algorithm 4: Local Search 1 (Hill Climb)

```
Data: A list of items N items and limit W
Result: A list of selected items
$current\_solution$ ←randomised N size list of 0's
  (not selected), 1's (selected)
$counter \leftarrow 0$
$runtime \leftarrow 0$
define $total\_value(list)$
    if $total\_weight(list) > W$ then
        return $-total\_weight(list) + capacity$
end
    return $total\_value$
while $counter \leq N^2$ & $runtime \leq cutoff$ do
    $neighbor \leftarrow current\_solution$
    $i, j \leftarrow random.sample(N, 2)$
    $neighbor[i] \leftarrow 1 - neighbor[i]$
    $neighbor[j] \leftarrow 1 - neighbor[j]$
    if $total\_value(neighbor) >$
      $total\_value(current\_solution)$ then
        $current\_solution \leftarrow neighbor$
    end
    $counter = counter + 1$
    update $runtime$
end
return $current\_solution$
```

(1) **Initialization:** The algorithm starts by randomly initializing a feasible solution to the Knapsack Problem. It iterates over each item in the list and adds it to the knapsack if adding the item does not exceed the capacity of the knapsack.
(2) **Temperature Settings:** The algorithm sets the initial temperature (`start_temp`) and the minimum temperature (`min_temp`). These parameters control the annealing schedule, which determines the rate at which the algorithm explores the solution space.
(3) **Iterative Improvement:**
   (a) The algorithm enters a loop that continues until the temperature drops below the minimum temperature threshold.
   (b) Within each iteration of the loop, the algorithm randomly selects an item (`candidate`) from the list of items.
   (c) If the selected item is already in the knapsack (`candidate.index in current_solution`), the algorithm considers removing it. If removing the item results in a feasible solution, it calculates the change in value (`delta_value`) and updates the solution if the change is positive or satisfies a probability condition based on the temperature (`random.random() < math.exp (delta_value / temperature)`).
   (d) If the selected item is not in the knapsack, the algorithm considers adding it. If adding the item results in a feasible solution, it calculates the change in value (`delta_value`) and updates the solution if the change is positive or satisfies a probability condition based on the temperature.
   (e) The temperature is reduced in each iteration according to the cooling schedule (`temperature *= alpha`), where `alpha` is a decay factor.
(4) **Solution Trace:** During the iterative process, the algorithm records the quality of the solution over time in a trace. This trace contains tuples of (`timestamp`, `current_value`), where `timestamp` represents the elapsed time since the start of the algorithm, and `current_value` represents the value of the current solution.
(5) **Termination:** The algorithm terminates when the temperature drops below the minimum temperature threshold.

Time and Space Complexity. Simulated Annealing requires constant space regardless of input size; therefore, space complexity is $O(1)$.

The time complexity is primarily determined by the number of iterations of the while loop, which depends on algorithm parameters and problem instance. Time complexity is approximately $O(\text{iterations})$.

5 EMPIRICAL EVALUATION

All algorithms were run with Python on an Intel i7-8650U CPU at 1.90GHz with a cache size of 8 MB, 16 GB of RAM, and Microsoft Windows 10 Pro.

A comprehensive table of all the results can be found in Figure 1. Below, we outline the specific results for each algorithm.

Branch and Bound: In terms of accuracy or relative error, Branch and Bound unsurprisingly performed the best out of all the algorithms. It found the optimal solution for all datasets except large_19, large_20, and large_21 where it had a relative error of 0.0022, 0.00065, and 0.00016 respectively. The algorithm was run with a

Algorithm 5: Local Search 2 (Simulated Annealing)

```
Data: List of items $N$, limit $W$, $start\_temp$,
      $alpha$, $min\_temp$
Result: Selected items list
Randomly initialize $current\_solution$,
  $current\_weight$, $current\_value$
Set $temperature$ to $start\_temp$
Initialize $trace$ as empty list
$start\_time \leftarrow$ current time
while $temperature > min\_temp$ do
    $candidate \leftarrow$ random item from $N$
    $delta\_value \leftarrow$ change in value if candidate
      is added/removed
    if $delta\_value > 0$ or random(0, 1)
      $< e^{\frac{delta\_value}{temperature}}$ then
        Update $current\_solution$,
          $current\_weight$, $current\_value$
        Append (current time −
          $start\_time$, $current\_value$) to $trace$
    end
    Update $temperature$
end
return $current\_value$, $current\_solution$, $trace$
```

cutoff time of 1,800 seconds, or 30 minutes, which is the reason the optimal solution wasn't found for the last few datasets.

Approximation: The approximation algorithm performed surprisingly well given its low theoretical guarantee of 1/2 the optimal solution. The highest relative error was 3.6% and was otherwise mostly lower then 1%. It also performed the fastest, being significantly faster than other approaches for larger datasets. All test scenarios completed in under 1 second, as such the cutoff time was not needed.

Local Search 1: Local search 1 performed notably poorly in large_15 and onwards, with relative errors up to 30%. Instead of limiting the runtime, dataset was run for N^2 iterations, because there are $O(N^2)$ possible combinations of one pair of elements in a list of N items.

The Qualified Runtime Distributions (QRTD) and the Solution Quality Distributions (SQD) of Local Search 1 for large_1 and large_3 data have been shown in Figure 2 to Figure 5. The QRTD plots show the probability that the algorithm will produce a solution of at least certain quality (Relative Errors of 10%, 20% and 30% in our case) as time progresses. The SQD plots show the probability of obtaining a particular quality solution for various run-times.

Similarly, Box Plots for both dataset have been shown in Figure 6 and Figure 7. The box plots are based on the run-time required to obtain a solution of at most 5% relative error.

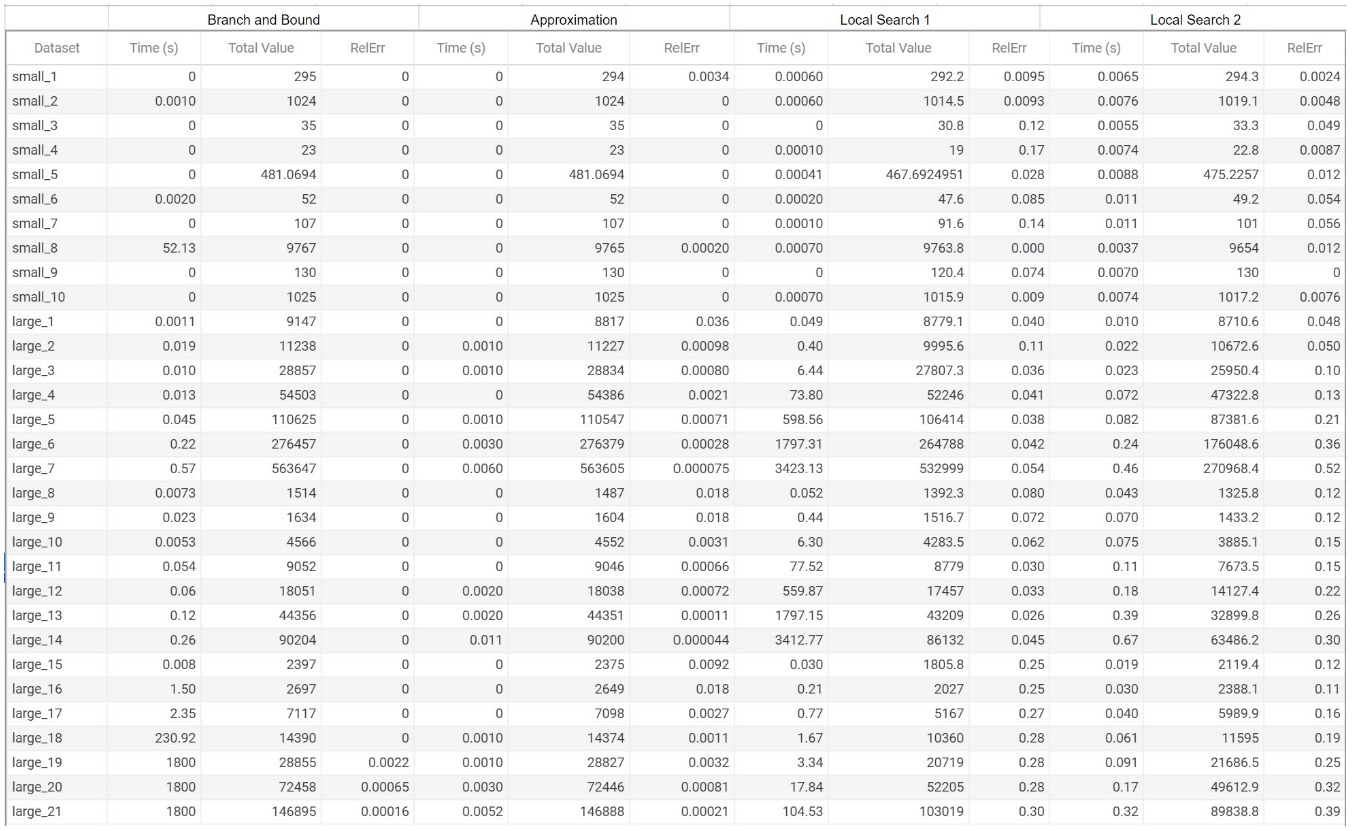
Local Search 2: LS2, which employing Simulated Annealing, exhibits strong performance on small datasets, achieving relative errors in the range of 0 to 0.05 and running times of 0 to 0.02 seconds. This highlights LS2's ability to produce quality results swiftly for small datasets. However, on large datasets, LS2's relative error varied from 0.05 to 0.5, with running times spanning from 0.01 to 0.70 seconds. The algorithm's iterative exploration process and temperature reduction to converge towards optimal solutions contribute to its slower and poor performance on larger datasets. Notably, LS2's performance underscores a trade-off between computational time and solution quality. The alpha decay factor, integral to the annealing process, significantly influences algorithm convergence. Proper tuning of parameters, including start and end temperatures and alpha, enables LS2 to deliver results swiftly, albeit at the expense of increased relative error for large datasets. This trade-off between time and accuracy highlights the nuanced nature of LS2's performance in empirical evaluations.

The Qualified Runtime Distributions (QRTD) and the Solution Quality Distributions (SQD) of Local Search 2 for large_1 and large_3 data have been presented in Figures 8 to 11. The QRTD plots depict the probability that the algorithm will produce a solution of at least certain quality (70%, 80%, and 90% of the optimal value) as time progresses, while the SQD plots illustrate the probability of obtaining a particular quality solution for various run-times.

Additionally, box plots for both datasets are shown in Figures 12 and 13, respectively. These box plots are based on the run-time required to obtain a solution with a relative error of at most 5%.

6 DISCUSSION

All four algorithms generally behave as expected based on their theoretical approaches. The branch and bound algorithm was more computationally expensive than the other algorithms for a number of tests, particularly



	Branch and Bound			Approximation			Local Search 1			Local Search 2		
Dataset	Time (s)	Total Value	RelErr	Time (s)	Total Value	RelErr	Time (s)	Total Value	RelErr	Time (s)	Total Value	RelErr
small_1	0	295	0	0	294	0.0034	0.00060	292.2	0.0095	0.0065	294.3	0.0024
small_2	0.0010	1024	0	0	1024	0	0.00060	1014.5	0.0093	0.0076	1019.1	0.0048
small_3	0	35	0	0	35	0	0	30.8	0.12	0.0055	33.3	0.049
small_4	0	23	0	0	23	0	0.00010	19	0.17	0.0074	22.8	0.0087
small_5	0	481.0694	0	0	481.0694	0	0.00041	467.6924951	0.028	0.0088	475.2257	0.012
small_6	0.0020	52	0	0	52	0	0.00020	47.6	0.085	0.011	49.2	0.054
small_7	0	107	0	0	107	0	0.00010	91.6	0.14	0.011	101	0.056
small_8	52.13	9767	0	0	9765	0.00020	0.00070	9763.8	0.000	0.0037	9654	0.012
small_9	0	130	0	0	130	0	0	120.4	0.074	0.0070	130	0
small_10	0	1025	0	0	1025	0	0.00070	1015.9	0.009	0.0074	1017.2	0.0076
large_1	0.0011	9147	0	0	8817	0.036	0.049	8779.1	0.040	0.010	8710.6	0.048
large_2	0.019	11238	0	0.0010	11227	0.00098	0.40	9995.6	0.11	0.022	10672.6	0.050
large_3	0.010	28857	0	0.0010	28834	0.00080	6.44	27807.3	0.036	0.023	25950.4	0.10
large_4	0.013	54503	0	0	54386	0.0021	73.80	52246	0.041	0.072	47322.8	0.13
large_5	0.045	110625	0	0.0010	110547	0.00071	598.56	106414	0.038	0.082	87381.6	0.21
large_6	0.22	276457	0	0.0030	276379	0.00028	1797.31	264788	0.042	0.24	176048.6	0.36
large_7	0.57	563647	0	0.0060	563605	0.000075	3423.13	532999	0.054	0.46	270968.4	0.52
large_8	0.0073	1514	0	0	1487	0.018	0.052	1392.3	0.080	0.043	1325.8	0.12
large_9	0.023	1634	0	0	1604	0.018	0.44	1516.7	0.072	0.070	1433.2	0.12
large_10	0.0053	4566	0	0	4552	0.0031	6.30	4283.5	0.062	0.075	3885.1	0.15
large_11	0.054	9052	0	0	9046	0.00066	77.52	8779	0.030	0.11	7673.5	0.15
large_12	0.06	18051	0	0.0020	18038	0.00072	559.87	17457	0.033	0.18	14127.4	0.22
large_13	0.12	44356	0	0.0020	44351	0.00011	1797.15	43209	0.026	0.39	32899.8	0.26
large_14	0.26	90204	0	0.011	90200	0.000044	3412.77	86132	0.045	0.67	63486.2	0.30
large_15	0.008	2397	0	0	2375	0.0092	0.030	1805.8	0.25	0.019	2119.4	0.12
large_16	1.50	2697	0	0	2649	0.018	0.21	2027	0.25	0.030	2388.1	0.11
large_17	2.35	7117	0	0	7098	0.0027	0.77	5167	0.27	0.040	5989.9	0.16
large_18	230.92	14390	0	0.0010	14374	0.0011	1.67	10360	0.28	0.061	11595	0.19
large_19	1800	28855	0.0022	0.0010	28827	0.0032	3.34	20719	0.28	0.091	21686.5	0.25
large_20	1800	72458	0.00065	0.0030	72446	0.00081	17.84	52205	0.28	0.17	49612.9	0.32
large_21	1800	146895	0.00016	0.0052	146888	0.00021	104.53	103019	0.30	0.32	89838.8	0.39

Figure 1: Comprehensive table of results for each algorithm on all small and large datasets.

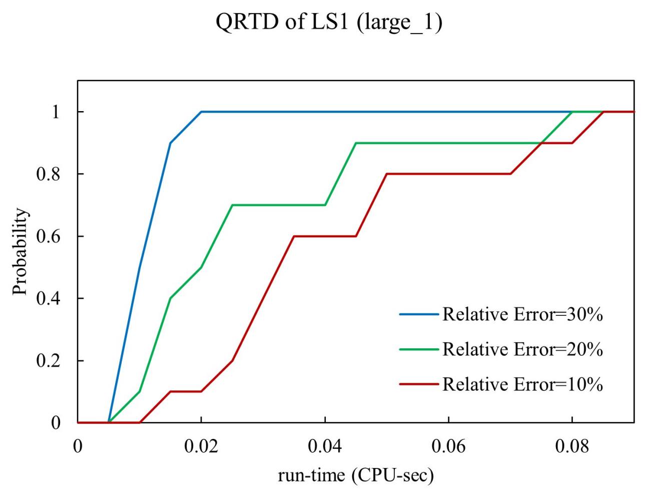


Figure 2: QRTD of large_1 with Local Search 1

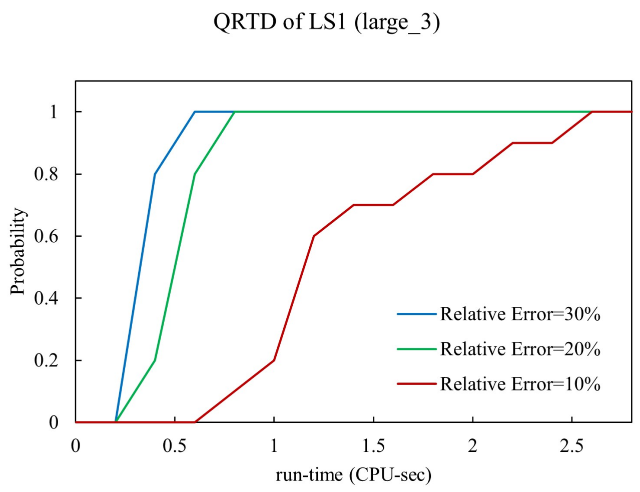


Figure 3: QRTD of large_3 with Local Search 1

larger datasets. For some, it had to rely on the approximation algorithm as a starting point to finish. The approximation algorithm performed below 4% error for all tested sets. The local search algorithms performed relatively well, but generally slightly worse than the approximation algorithm. This was somewhat counter

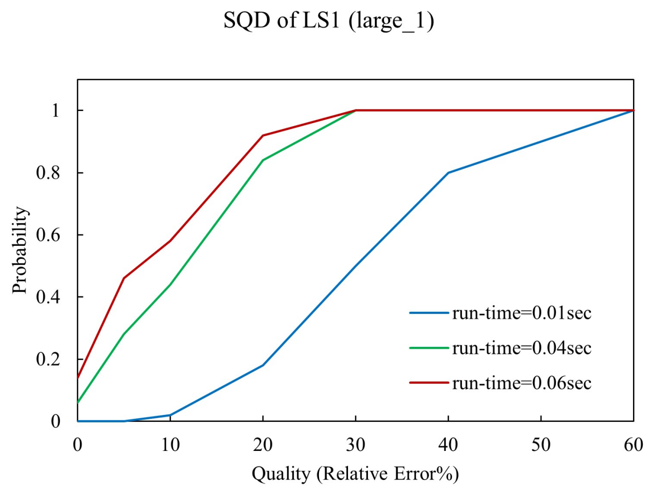


Figure 4: SQD of large_1 with Local Search 1.

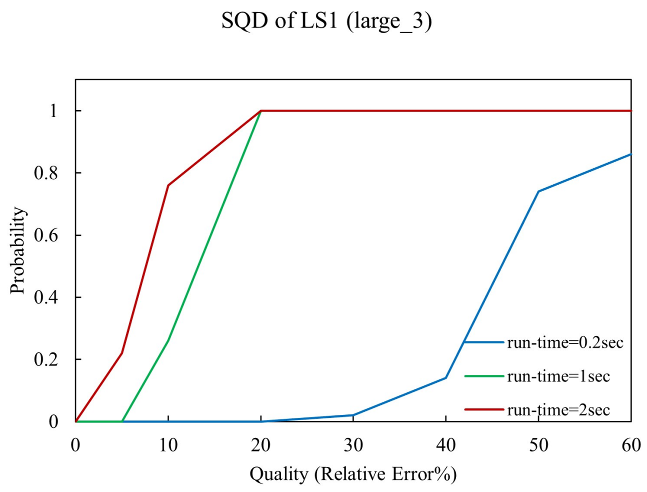


Figure 5: SQD of large_3 with Local Search 1

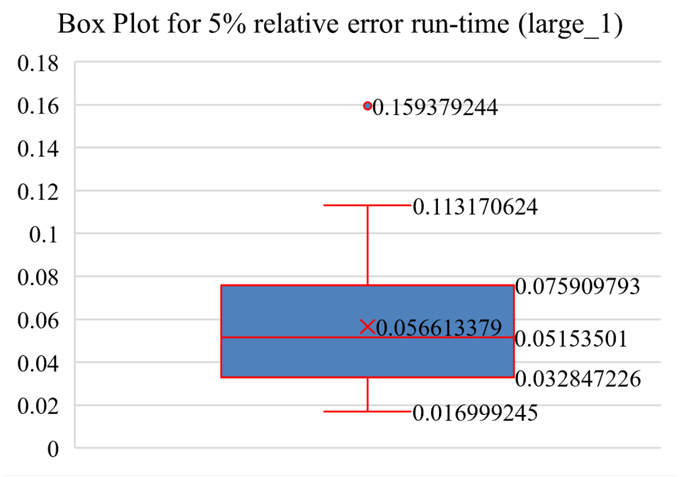


Figure 6: Box Plot of large_1 with Local Search 1.

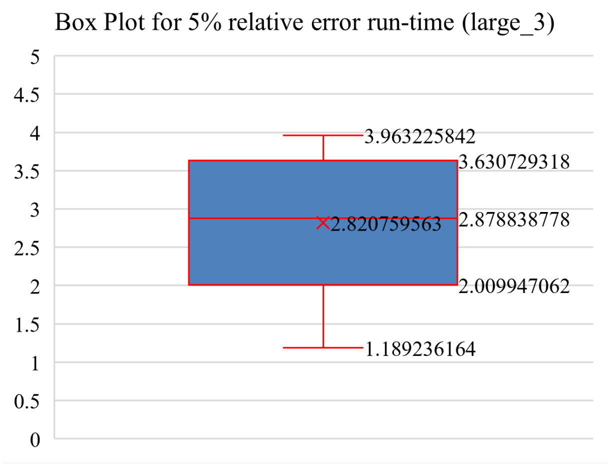


Figure 7: Box Plot of large_3 with Local Search 1

to the expectation that local search usually performs closer to optimal than construction heuristics, though with no guarantees.

The algorithms' run-times fit the expected results based on their time complexities. The approximation algorithm performed the fastest, while retaining relatively high accuracy. There were a few interesting results, including small_8 for branch and bound, which took 52 seconds despite the small dataset size.

The empirical evaluation of the implemented algorithms reveals distinct trade-offs between computational efficiency and solution quality across different problem instances. Branch and Bound (BnB) consistently demonstrated superior accuracy, achieving optimal solutions for the majority of datasets, albeit with longer runtimes. The algorithm's exhaustive search strategy ensures optimality but incurs higher computational costs, particularly evident in larger instances.

In contrast, the Approximation algorithm showcased remarkable efficiency, achieving relatively low relative errors within significantly shorter runtimes. Despite its lower theoretical guarantee, the algorithm's greedy approach produced satisfactory results, especially notable for larger datasets where it outperformed other

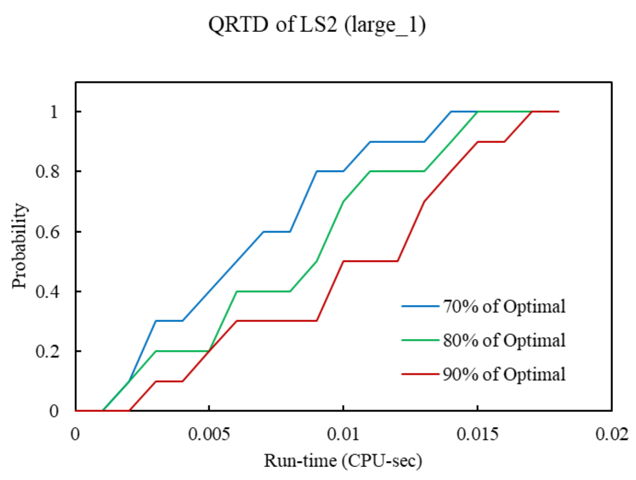


Figure 8: QRTD of Large_1 with Local Search 2

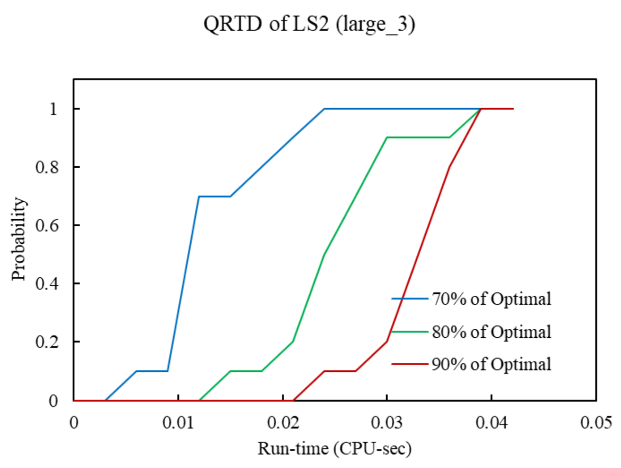


Figure 9: QRTD of Large_3 with Local Search 2

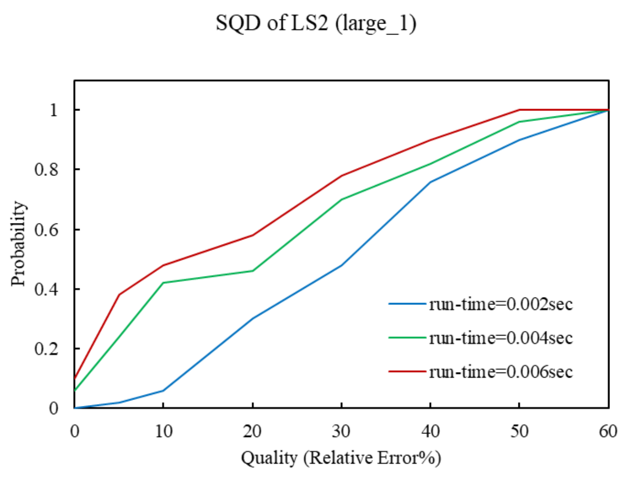


Figure 10: SQD of Large_1 with Local Search 2

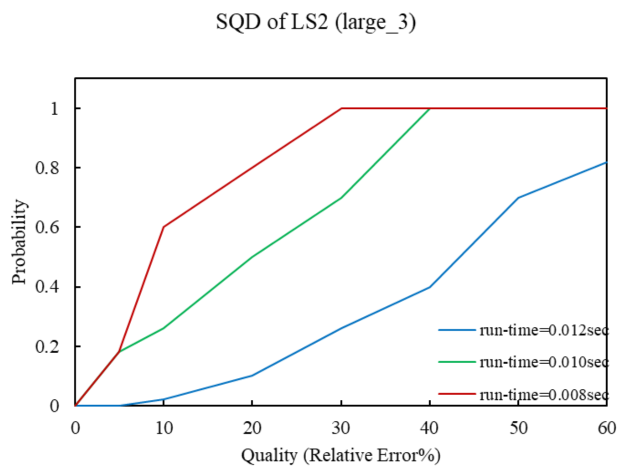


Figure 11: SQD of Large_3 with Local Search 2

methods in terms of speed. This highlights the effectiveness of leveraging heuristics to quickly approximate solutions, albeit with a potential sacrifice in optimality.

Local Search 1 (LS1), employing hill climbing, demonstrated mixed performance, particularly struggling with larger datasets where its runtime complexity became prohibitive. Despite attempts to mitigate this through iterative improvement strategies, LS1's scalability remains a challenge. The approach's dependence on exhaustive search limits its applicability to larger problem instances.

Local Search 2 (LS2), utilizing Simulated Annealing, exhibited a nuanced trade-off between computational time and solution quality. While effective for small datasets, LS2's performance deteriorated on larger instances due to the iterative exploration process. However, with careful parameter tuning, LS2 demonstrated the potential to yield acceptable solutions within shorter timeframes, underscoring the importance of optimization in algorithmic configuration.

Overall, the evaluation underscores the need for a tailored approach to algorithm selection based on problem characteristics and computational constraints. While

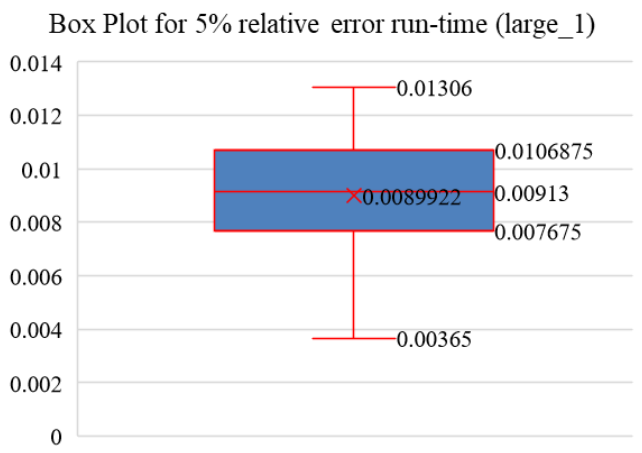


Figure 12: Box Plot of Large_1 with Local Search 2

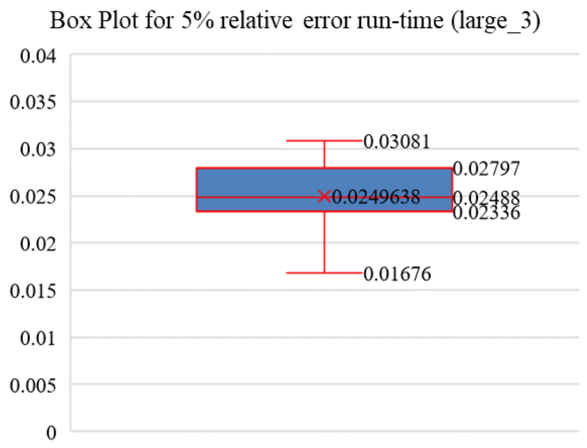


Figure 13: Box Plot of Large_3 with Local Search 2

BnB ensures optimality at the expense of computational resources, heuristics like Approximation and LS2 offer viable alternatives for scenarios where efficiency is paramount. Future research could focus on enhancing the scalability of local search methods and refining parameter optimization techniques to strike a better balance between time and accuracy across diverse problem domains.

7 CONCLUSION

Numerous approaches were taken to solve the Knapsack Problem, and their different advantages and disadvantages were investigated. In conclusion, the evaluation highlights the diverse performance of implemented algorithms. Branch and Bound (BnB) consistently achieves almost optimal accuracy but with longer runtimes, particularly for larger datasets. Approximation excels in efficiency, providing satisfactory results quickly despite lower guarantees. Local search methods show mixed performance, with scalability challenges for LS1 and nuanced trade-offs for LS2. Tailored algorithm selection is crucial, considering problem characteristics and computational constraints. Future research could focus on optimizing local search scalability and parameter tuning for improved efficiency and accuracy across problem domains, as well as using other upper bound functions for Branch and Bound that could improve its running time.

REFERENCES

[1] Daniel S Hirschberg and Chung K Wong. A polynomial-time algorithm for the knapsack problem with two variables. *Journal of the ACM (JACM)*, 23(1):147–154, 1976.

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

[3] Silvano Martello and Paolo Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1(3):169–175, 1977.

[4] Silvano Martello and Paolo Toth. Algorithms for knapsack problems. *North-Holland Mathematics Studies*, 132:213–257, 1987.

[5] Joachim H Ahrens and Gerd Finke. Merging and sorting applied to the zero-one knapsack problem. *Operations Research*, 23(6):1099–1109, 1975.

[6] Yalçın Akçay, Haijun Li, and Susan H Xu. Greedy algorithm for the general multidimensional knapsack problem. *Annals of operations research*, 150:17–29, 2007.

[7] Nima Moradi, Vahid Kayvanfar, and Majid Rafiee. An efficient population-based simulated annealing algorithm for 0–1 knapsack problem. *Engineering with Computers*, 38(3):2771–2790, 2022.

[8] Saïd Hanafi and Arnaud Freville. An efficient tabu search approach for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 106(2-3):659–675, 1998.