CSE6140 Group Project: Knapsack Problem

Richard Zhi Zhao
rzhao97@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Yuang Zhang
yzhang3992@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Chenyu Liu
cliu797@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Chengqian Zhang
czhang657@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

1 Introduction

The Knapsack Problem is a classic combinatorial optimization problem with significant theoretical and practical importance. Given a set of items, each with a weight and value, the objective is to determine which items to include in a knapsack so that the total value is maximized, subject to a weight capacity constraint. This NP-complete problem arises in many real-world domains such as resource allocation, financial portfolio optimization, and cargo loading.

In this project, we implemented and evaluated several algorithmic approaches to the 0-1 Knapsack Problem, including an exact branch-and-bound method, an approximation algorithm with theoretical guarantees, and two local search heuristics. Through empirical analysis on benchmark datasets of varying size and structure, we assessed the trade-offs between solution quality and computational efficiency across the different techniques. Our goal was to gain insights into the strengths and limitations of each algorithmic paradigm for solving the Knapsack Problem in practice.

2 Problem Definition

An instance of the 0-1 Knapsack Problem is defined by:

- A set of n items, indexed from 1 to n
- For each item i, a non-negative weight w_i and value v_i
- A positive knapsack capacity W

The binary decision variables $x_i \in 0, 1$ indicate whether item i is selected for the knapsack ($x_i = 1$) or not ($x_i = 0$). Using this notation, the 0-1 Knapsack Problem can be formulated as the following integer linear program:

Maximize $\sum_{i=1}^{n} v_i \cdot x_i$

Subject to:

$$\sum_{i=1}^{n} w_i \cdot x_i \leq W$$

$$x_i \in 0, 1 \quad \text{for } i = 1, \ldots, n$$

The objective is to find a subset of the items that maximizes the total value while respecting the knapsack's weight capacity. The constraint ensures the total weight of selected items does not exceed W, and the binary restrictions on x_i enforce selecting items in their entirety or not at all.

3 Related Work

The Knapsack Problem has been extensively studied since the pioneering work of Dantzig in the 1950s [1]. The problem and its variants are the subject of several surveys, including Kellerer et al. and Martello et al. [2, 3]. Exact algorithms based on dynamic programming, branch-and-bound, and hybrid approaches have been proposed. Toth and Horowitz and Sahni developed early branch-and-bound methods leveraging upper bounds from the linear relaxation [4, 5]. More recently, Pisinger introduced a series of increasingly complex branch-and-bound algorithms incorporating dominance relations, variable reduction, and other enhancements [6]. However, given the problem's NP-completeness, such methods become impractical for large instances.

To attain more efficient solutions, approximation algorithms and heuristics are often employed. A simple greedy 2-approximation scheme that sorts items by value-to-weight ratio was outlined by Dantzig [1]. This was generalized to a polynomial-time approximation scheme (PTAS) by Ibarra and Kim [7]. Sahni later devised an efficient fully polynomial-time approximation scheme (FPTAS) [8].

A variety of local search techniques have also been adapted for the Knapsack Problem, including simulated annealing, tabu search, and genetic algorithms [9–11]. The choice of solution representation, neighborhood structure, and search strategies heavily influences the performance of these heuristics.

Our work builds upon this rich literature by empirically comparing exact, approximate, and heuristic algorithms using a common experimental framework. This enables us to assess how theoretical differences translate into solution quality and scalability in practice on both real and generated instances. We also introduce some novel implementation choices within the local search paradigm.

4 Algorithms

In this project, we implemented three categories of algorithms to solve the 0-1 Knapsack Problem: an exact algorithm, an approximation algorithm, and two local search heuristics. Each category represents a distinct approach to tackling the problem, offering different trade-offs between solution quality and computational efficiency.

4.1 Exact Algorithm

We implemented a branch-and-bound algorithm as our exact method for solving the Knapsack Problem. Branch-and-bound is a general algorithmic technique that explores the solution space by systematically enumerating candidate solutions in a search tree. The algorithm maintains upper and lower bounds on the optimal solution value, using them to prune subproblems that cannot yield a better solution than the current best. However, while this exact approach guarantees finding an optimal solution, its worst-case time

complexity remains exponential, limiting its practicality for large instances.As a consequence, the "large_scale" file costs us much time to get the computational result. Here is the peusdo-code for the branch-and-bound algorithm:

```
Algorithm BnBKnapsack(items, capacity):
    Sort items by value/weight DESC
    Priority queue Q
    MaxProfit to 0
    BestItems to []

    Initialize a root node with level -1, value 0, weight 0, and no items
    Q = Q + {root Node}

    while Q is not a empty Priority Queue:
        u = Q.pop()

        if u.level == N - 1
            continue
        end if

        Node V to handle the case that the next item is included:
            v.level = u.level + 1
            v.weight = u.weight + items[v.level].weight
            v.value = u.value + items[v.level].value

        if v.weight <= capacity:
            if v.value > maxProfit:
                maxProfit = v.value
                bestItems = v.items
            end if
            if Bound(v) > maxProfit:
                Q = Q + {v}
            end if
        end if

        Node w to handle the case that the next item is not included:
            w.level = u.level + 1
            w.weight = u.weight
            w.value = u.value

        if Bound(w) > maxProfit:
            Q = Q + {w}
        end if
    end while

    return maxProfit, bestItems

Function Bound(node)
    if node.weight >= capacity:
        return 0
    end if

    bound = node.value
    j = node.level + 1
    totweight = node.weight

    while j < N and totweight + items[j].weight <= capacity:
        totweight += items[j].weight
        bound += items[j].value
        j += 1
    end while

    if j < N:
        bound += (capacity - totweight) * items[j].value / items[j].weight
    end if

    return bound
```

4.2 Approximation Algorithm

To obtain more efficient solutions with provable quality guarantees, we implemented an approximation algorithm for the Knapsack Problem. This algorithm offers a compelling alternative to exact methods for large-scale instances where near-optimal solutions suffice.

For our implementation of the Approximation Algorithm, we have used a version of the greedy algorithm in order to achieve a 2-approximation of the optimal solution. It performs this by sorting each data entry by a "value density" from highest to lowest, which will be calculated by taking the value of each entry and dividing it by the weight. The algorithm passes through the sorted list from index 0, while maintaining a running count of accumulated data points, weight, and value. At each passed point, it checks if the weight of that data point plus the weight of all already selected points is greater than the max weight. If it is not, then the algorithm records the index of the point, and adds its weight to the weight total and the value to the accumulated value. If it is, the algorithm simply exits and no longer passes through the list (though it makes a final check to see if the value of the next element would exceed the total value of the currently selected elements, to deal with edge cases where a single element contains over half of the total possible value). At the end, it can output this accumulated value and selected points as its output.

To prove this is a 2-approximation, we can consider the first element in the sorted list whose weight would cause the accumulated weight to exceed the total weight (If this never occurs, then the optimal solution is to take every element, which the Greedy Algorithm can and would do). Let us say that this occurs at index j. Since we have sorted by weight, and we know that the weight of element $j + 1$ would cause the knapsack to overflow, we know that the total value of this modified Greedy solution plus the value of element $j + 1$ will exceed the total value of the optimal solution. Therefore, if we denote v_i as the value of index i in the sorted array, v_G as the Greedy Solution, and v^* as the optimal solution, we can write that:

$$v_G = \sum_{i=0}^{k} v_i \leq v^* < \sum_{i=0}^{k+1} v_i < 2v_G$$

Therefore, we have a 2-approximation.

Below is the pseudocode for the algorithm:

```
def greedy(numItems, maxWeight, [weight, value]):
    [weight, value] = sort([weight, value], key=(value/weight))
    totalWeight = 0
    totalValue = 0
    items = []

    for item in [weight,value]:
        totalWeight = totalWeight + item[weight]
        if totalWeight > maxWeight:
            return items, totalValue
        totalValue = totalValue + item[value]
        items.append(item)

    return items, totalValue
```

The sorting algorithm will run in approximately $O(nlogn)$ time. The loop itself runs in just n time, as it is a single loop across all

elements through all items. Therefore, the total runtime of the algorithm is $O(nlogn + n) = O(nlogn)$. The space complexity of the algorithm is solely based on the item to store the currently selected items, as no other data structures are created. As such, it has a space complexity of approximately $O(n)$.

The benefit of this algorithm is that it is very fast compared to other methods. The runtime is dominated by the necessity of a sorted dataset instead of the algorithm itself, and it also places a very small memory load as the most elements it has to store is just the size of the dataset itself. Of course, these time and space benefits are compensated for by the accuracy limitations of the algorithm. We are forced to give a bound with no controls on accuracy like an FPTAS or PTAS solution could potentially be able to provide.

4.3 Local Search

Local search is a broad class of heuristic optimization techniques that iteratively refine a candidate solution by exploring its neighborhood in the solution space. We implemented two distinct local search algorithms for the knapsack problem.

Simulated Annealing Simulated Annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It introduces a "temperature" parameter that allows the algorithm to escape local optima by accepting worse solutions with a decreasing probability. This characteristic makes it suitable for more complex landscapes. The algorithm's probabilistic nature and cooling schedule enable it to adapt to various problem structures, making it versatile for different optimization scenarios. Simulated Annealing's ability to accept worse solutions provides a mechanism for broader exploration, allowing it to discover solutions that Hill Climbing might miss. The cooling rate and initial temperature provide flexibility in controlling the algorithm's behavior, allowing for fine-tuning based on problem characteristics. However, Simulated Annealing is more complex to implement and requires careful tuning of parameters, such as temperature and cooling rate, to achieve optimal results. Its probabilistic elements may produce varying results for the same input, requiring multiple runs to ensure robustness. The performance of Simulated Annealing depends heavily on the correct tuning of parameters. Inappropriate settings can lead to poor convergence or excessive running times. Given n items, choosing a random neighbor has $O(n)$ complexity, while the total complexity is influenced by the number of iterations, temperature, and cooling rate. The space complexity for Simulated Annealing is also generally $O(n)$, similar to Hill Climbing, as it maintains the current solution, best solution, and trace file.Additional space might be required for temperature management and acceptance probabilities. Simulated Annealing was selected for its ability to escape local optima, a common issue with Hill Climbing. The algorithm introduces a temperature parameter that allows it to accept worse solutions with a certain probability, promoting exploration and reducing the risk of stagnation.

Simulated Annealing's key components include:

Temperature Schedule: The initial temperature is set to 100, allowing for significant exploration and acceptance of worse solutions.

Final Temperature: The final temperature is set to 1, indicating when the algorithm should stop.

Acceptance Probability: Higher temperatures lead to a greater acceptance probability of worse solutions.

Cooling Rate: A cooling rate of 0.99 was chosen to gradually reduce the temperature, balancing exploration and exploitation.

Below is the Pseudocode for Simulated Annealing:

```
def sa(values, weights, capacity, temp_init, temp_final, alpha,
cutoff, seed):
    n = len(values)
    sol = generate_solution(n)
    val, wt = compute_value_weight(sol, values, weights)

    while wt > capacity:
        idx = select_random(sol)
        sol[idx] = 0
        val -= values[idx]
        wt -= weights[idx]

    best_sol = sol[:]
    best_val = val
    temp = temp_init
    start = get_time()

    while temp > temp_final and within_time(cutoff, start):
        idx = random_int(0, n - 1)
        new_sol = flip(sol, idx)
        new_val, new_wt = compute_value_weight(new_sol, values,
        weights)

        if new_wt > capacity:
            continue
        delta = new_val - val
        accept_prob = calc_acceptance(delta, temp)

        if random_float() < accept_prob:
            sol = new_sol
            val = new_val
            wt = new_wt
            if val > best_val:
                best_sol = sol[:]
                best_val = val
                update_trace(solution_trace, best_val, start)
        temp *= alpha
    return best_sol, best_val, solution_trace
```

Hill Climbing Hill Climbing is a local search algorithm that starts with an initial solution and iterates through neighboring solutions to find improvements. The algorithm is designed to move toward better solutions by making incremental changes, with each iteration exploring neighboring states. Hill Climbing is straightforward to implement and understand, making it an attractive choice for smaller-scale problems or quick prototyping. Besides, in scenarios where a gradient-like approach is effective, Hill Climbing can quickly converge to a local optimum. This can be advantageous when the problem has a relatively simple solution landscape. Moreover, hill Climbing usually follows a deterministic path, leading to reproducible results when run with the same initial conditions and parameters. However, it tends to get stuck in local optima, especially when the solution landscape has multiple peaks or plateaus. This limitation can significantly impact the quality of the solution. Hill Climbing does not inherently include mechanisms for escaping local optima, making it less effective in highly complex or multimodal landscapes. The algorithm's focus on neighboring solutions can limit its ability to explore the broader solution space, leading to suboptimal results. The quality of the solution can be highly dependent on the initial solution, which can vary with different random seeds. The time complexity of Hill Climbing

depends on the number of iterations and the number of neighbors explored at each iteration. Given n items, exploring each neighbor in a brute-force manner would have a complexity of $O(n)$ per iteration. The total complexity is influenced by the number of iterations and can vary depending on the search space and convergence rate. The space complexity for Hill Climbing is generally $O(n)$, as it maintains the current solution and neighboring solutions. Additionally, space is used to track the best solution and the trace file for recording significant improvements.

Hill Climbing's core components include:

Initial Solution: The initialization method involves randomly assigning each item a value of 0 or 1, providing diversity in the starting point.

Neighborhood Structure: The algorithm explores neighboring solutions by flipping one item at a time.

Stopping Criteria: Hill Climbing stops when no better neighbors are found or when a specified time limit is reached.

Below is the Pseudocode for Hill Climbing:

```
def hill_climbing(values, weights, capacity, cutoff):
    n = len(values)
    sol = generate_solution(n)
    val, wt = compute_value_weight(sol, values, weights)

    while wt > capacity:
        idx = select_random_item(sol)
        sol[idx] = 0
        val -= values[idx]
        wt -= weights[idx]

    best_sol = sol[:]
    best_val = val
    trace = initialize_trace(best_val)
    start = get_time()

    while within_time(cutoff, start):
        best_neighbor = find_best_neighbor(sol, values, weights,
        capacity, n)
        if not best_neighbor:
            break
        update_solution(sol, val, best_neighbor)
        if best_neighbor["value"] > best_val:
            best_sol = best_neighbor["solution"][:]
            best_val = best_neighbor["value"]
            update_trace(trace, best_val, start)

    return best_sol, best_val, trace
```

5 Empirical Evaluation

We conducted our experiments on a machine with an AMD Ryzen 7 7840HS 3.80GHz CPU, 16.0 GB RAM, using Python 3.10.11 and Visual Studio Code (version 1.88) on Windows 11 64-bit OS. For each algorithm implementation, we ran the code on all provided Knapsack problem instances - the 10 small scale instances, and 21 large scale instances.

For the branch-and-bound and approximation algorithms, we report the total knapsack value obtained, relative error compared to the optimum, and running time. For the local search algorithms LS1 and LS2, we ran each instance 10 times with different random seeds and report the average total value, average relative error, and average running time. The comprehensive results are shown in Table 1.

From the results, we see that the Branch-and-Bound algorithm finds the optimal solutions in all instances, but it takes significantly more time on larger cases, most of them even exceeds 1 hour. The approximation algorithm runs extremely fast, finishing even the largest instance in under 0.01 second, and on most instances achieves a relative error of under 1%. Our local search algorithm 1 (simulated annealing) also runs fast, and has lower relative error on small scale instances comparing to approximation algorithm, but the relative error becomes bigger on larger instances. The running time of local search algorithm 2 (hill climbing) is larger than simulated annealing and approximation, but smaller than the branch-and-bound, however, it has a much greater relative error, with most instances achieve a relative error of higher than 30%. Overall, the approximation algorithm struck a balance between solution quality and speed, obtaining near-optimal solutions in less than a second.

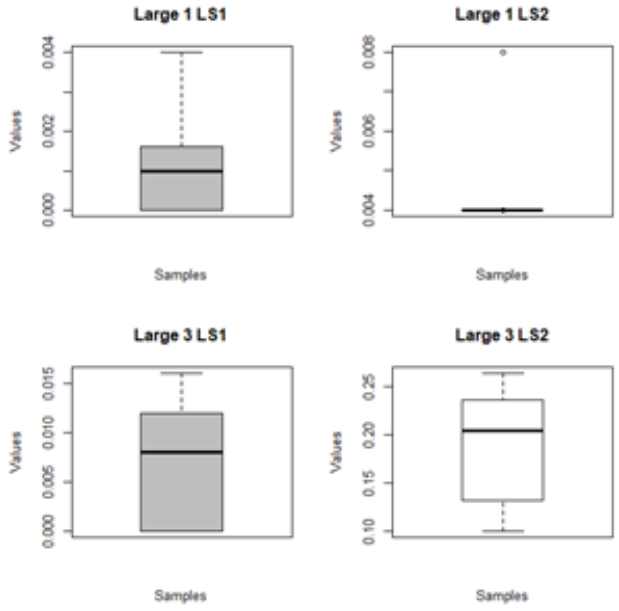

Figure 1: Boxplots for LS running time

The box plots illustrate the running time of two local search algorithms, Simulated Annealing and Hill Climbing, across two datasets, "large 1" and "large 3". For "Large 1 LS1", the box plot shows a narrow interquartile range with a median skewed towrds the lower quartile, indicating a concentrated distribution with limited variability. No significant outliers are observed, suggesting consistent performance from this algorithm setup. In contrast, "Large1 LS2" presents a highly skewed distribution with a low IQR and outliers. The box plot for "Large 3 LS1" exhibits a wider IQR, indicating greater variability in running time. The presence of mild outliers points to some level of variability in performance, possibly due to different initial conditions or varying solution landscapes. Finally, the box plot for "Large LS2" displays a higher median with broader IQR, suggesting a more significant range of running time.

6 Discussion

The results demonstrate the trade-offs between exact, approximate, and heuristic approaches for the knapsack problem. The Branch-and-Bound algorithm guarantees optimality but scales poorly to

Dataset	Branch and Bound			Approximation			Local Search 1			Local Search 2		
	Value	Time(s)	RelErr	Value	Time(s)	RelErr	Value	Time(s)	RelErr	Value	Time(s)	RelErr
small_1	295	0.00	0	290	0.00055	0.016	291.8	0	0.01	231.7	0	0.21
small_2	1024	0.03	0	1018	0.00099	0.0059	993.3	0.001	0.03	979.7	0.001	0.04
small_3	35	0.00	0	35	0.001	0	35	0	0	31.4	0	0.1
small_4	23	0.00	0	16	0.001	0.30	23	0	0	21	0	0.09
small_5	481.07	0.00	0	481.07	0.001	0	463.6	0.001	0.04	329.2	0	0.32
small_6	52	0.00	0	52	0	0	51.5	0.001	0.01	47.3	0.0004	0.09
small_7	107	0.00	0	90	0	.16	106.8	0	0	91.1	0.001	0.15
small_8	9767	291.92	0	9266	0	0.051	9690.9	0	0.01	9688.9	0	0.01
small_9	130	0.00	0	130	0	0	130	0	0	124.3	0	0.04
small_10	1025	0.02	0	1019	0.01	0.0059	993.1	0	0.03	974.5	0.03	0.05
large_1	9147	0.01	0	8817	0.001	0.036	4050.2	0.002	0.56	2473.6	0003	0.73
large_2	11238	0.03	0	11227	0	0.00098	4379.2	0.005	0.61	4397.7	0.027	0.61
large_3	13658.0	30.15	0	28834	0.0011	0.00080	7852.7	0.005	0.73	6045.1	0.199	0.79
large_4	13920.0	30.15	0	54046	0.00092	0.0084	12076.3	0.025	0.78	9457.8	1.019	0.83
large_5	13802.0	30.13	0	110328	0.0011	0.0027	17459.3	0.023	0.84	14214.7	3.803	0.87
large_6	11529.0	30.07	0	276371	0.0040	0.00031	38953.7	0.205	0.86	29873.3	26.79	0.89
large_7	9542.0	30.04	0	563534	0.008	0.0002	60584.8	0.353	0.89	54468.4	113.92	0.9
large_8	1514.0	0.00	0	1276	0	0.16	1124.9	0.003	0.26	1054.1	0.003	0.3
large_9	1634.0	0.01	0	1463	0	0.10	1144.5	0.008	0.3	1113.9	0.016	0.32
large_10	2565.0	30.21	0	4551	0.001	0.0033	2893.4	0.029	0.37	2700.3	0.105	0.41
large_11	1850.0	30.18	0	9046	0.00098	0.00066	5675	0.05	0,37	5231.4	0.476	0.42
large_12	1571.0	30.14	0	17834	0.012	0.0020	10709.3	0.15	0.41	10080.2	2.325	0.44
large_13	1209.0	30.09	0	44238	0.0040	0.0026	26102.1	0.49	0.41	25288.9	11.419	0.43
large_14	1055.0	30.05	0	90172	0.0087	0.00035	51311	1.038	0.43	48983.33	58.284	0.45
large_15	2397.0	0.04	0	2375	0.0010	0.0091	1579.5	0.005	0.34	1381.9	0.004	0.42
large_16	2697.0	0.69	0	2649	0	0.018	1642	0.004	0.39	1264.6	0.021	0.53
large_17	2296.0	30.54	0	7098	0	0.0027	3794.1	0.02	0.47	3046.1	0.102	0.57
large_18	2101.0	30.22	0	14374	0.0020	0.0011	6869.4	0.028	0.52	5949.3	0.29	0.59
large_19	2027.0	30.10	0	28827	0.0020	0.0032	12718.5	0.062	0.56	11749	1.284	0.59
large_20	1841.0	30.06	0	72446	0.0040	0.00081	30975	0.113	0.57	29785	8.636	0.59
large_21	1717.0	30.03	0	146888	0.0075	0.00021	60349	0.179	0.59	59046.6	36.469	0.6

Table 1: Algorithm Results on Knapsack Instances

large instances due to its exponential worst-case quality. For local search algorithms, simulated annealing generally outperformed hill climbing, finding better solutions in less time, especially for large instances, but they both has a relatively bad correctness of solution for larger instances, also indicating room for improvement in terms of scalability. The approximation algorithm is extremely fast and also keeps a low relative error, provide high-quality solutions in reasonable time by exploring the solution space efficiently.

7 Conclusion

In this project, we implemented and empirically evaluated exact branch-and-bound, approximation, and local search algorithms for the 0-1 knapsack problem. Our experiments on a diverse set of benchmarks revealed the strengths and limitations of each approach.

The branch-and-bound algorithm is suitable for small to medium-sized instances where optimamlity is crucial. The approximation algorithm is ideal when both speed and solution quality are required at a relative high level. The local search algorithms, especially simulated annealing, delivers near optimal solutions quickly for small instances. However, further research is needed to improve the scalability of local search on larger instances.

The project provide valuable insights into algorithm design, implementation, and performance analysis for a fundamental combinatorial optimization problem. The concepts and techniques learned can be extended to other domains where balancing solution quality, running time, and instance size is critical. Future work could explore more advanced local search strategies, parallelization, and hybrid algorithms to push the boundaries of what is achieved in practice.

References

[1] George B Dantzig. Discrete-variable extremum problems. *Operations research*, 5(2):266–288, 1957.

[2] Hans Kellerer, Ulrich Pferschy, David Pisinger, Hans Kellerer, Ulrich Pferschy, and David Pisinger. Introduction to np-completeness of knapsack problems. *Knapsack problems*, pages 483–493, 2004.

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

[4] Paolo Toth. Dynamic programming algorithms for the zero-one knapsack problem. *Computing*, 25(1):29–45, 1980.

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

[6] David Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45(5):758–767, 1997.

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

[8] Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 22(1):115–124, 1975.

[9] Andreas Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40(1):1–8, 1988.

[10] Frank Dammeyer and Stefan Voß. Dynamic tabu list management using the reverse elimination method. *Annals of Operations Research*, 41(2):29–46, 1993.

[11] Paul C Chu and John E Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4:63–86, 1998.