0-1 Knapsack Problem Exploration

Seth Nye
Georgia Institute of Technology
Atlanta, Georgia, USA

Mark Glinberg
Georgia Institute of Technology
Atlanta, Georgia, USA

Shih-Huan Chou
Georgia Institute of Technology
Atlanta, Georgia, USA

Steve Tang
Georgia Institute of Technology
Atlanta, Georgia, USA

ACM Reference Format:
Seth Nye, Mark Glinberg, Shih-Huan Chou, and Steve Tang. 2024. 0-1 Knapsack Problem Exploration. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The problem that is being explored in this project is the 0-1 knapsack problem. This problem can be defined as finding a way to select a subset of items with weights and values so as to maximize the value of items in the subset without exceeding a specified weight threshold.

In order to solve this problem, our team explored four different types of algorithms: branch and bound, approximation using a greedy method, local search using hill climbing, and local search using simulated annealing. From our experiments, we have found that the 0-1 knapsack problem is simple to solve with small input sizes. However, as the input size becomes large, our algorithms begin to perform more poorly. In general, the approximation algorithm performs well creating a solid solution quickly. However, it does not achieve the optimal solution in most cases regardless of input size. In contrast, our branch and bound algorithm always finds the optimal solution but is incredibly slow since it iterates over all possible solution states. Our local search algorithms both perform well for small input sizes. However, for large input sizes both local search algorithms see an increase in relative error with hill climbing performing worse on average compared to simulated annealing. Overall, our exploration of these algorithms demonstrates that the 0-1 knapsack problem can be solved with various approaches. However, we find there are some approaches that perform better than others depending on the data provided.

2 PROBLEM DEFINITION

For a set of n items, each item, i, has a nonnegative weight w_i and a nonnegative value v_i. The goal is to find a subset of items S so that the total weight of the items $\sum_{i \in S} w_i$ is less than the weight limit W and the total value of items chosen $\sum_{i \in S} v_i$ is maximized. Formally, this problem can be defined as follows given a set S' containing possible solutions as subsets.

$$\begin{gathered} max_{S \in S'}(\sum_{i \in S} v_i), S \subset S' \\ \text{Subject to} \sum_{i \in S} w_i \leq W \end{gathered} \tag{1}$$

3 RELATED WORK

Much of the existing work related to the 0-1 knapsack problem consists of the comparative analysis of various algorithms. One algorithm that is know to always solve this problem is the brute-force approach. However, it is also shown that this algorithm is incredibly inefficient requiring $O(2^n)$ time complexity since it iterates over each combination [4].

The next approach to consider is a greedy approach. Typically, there are 3 greedy approaches to consider: filling the knapsack by greatest value, filling the knapsack by lowest weight, and filling the knapsack by ratios of value to weight in non-increasing order [2]. All 3 approaches are unable to achieve the optimal solution. However, the third can get reasonably close in a short amount of time using $O(NlogN)$ time complexity [3]. In our experimentation, we use the 3rd greedy algorithm as an approximation algorithm to achieve a relatively strong solution in a short amount of time.

Another approach is a dynamic programming approach. This approach uses a 2-D table of size $O(nW)$ to solve subproblems where n is the number of items and W is the weight capacity [4]. In this approach, the subproblems are given by the remaining capacity of the knapsack after adding item i and the items from 1 to $i-1$. The solution in this case is given by a set S containing items chosen. The recurrence relation for this approach can be given as follows [4]:

$$V[n, W] = \begin{cases} 0, & \text{if } n = 0 \text{ or } W = 0. \\ V[n-1, W], & \text{if } \sum_{w \in S} > W. \\ max(v[n] + v[n-1, W - w_n], V[n-1, W]), & \text{if } n > 0 \text{ and } \sum_{w \in S} \leq W. \end{cases} \tag{2}$$

This algorithm is capable of finding the optimal solution using $O(nW)$ space complexity and $O(nW)$ time complexity. As such, dynamic programming is a strong approach to solve the 0-1 knapsack problem.

Another way to approach the 0-1 knapsack problem is local search algorithms. One of the most common local search approaches is a genetic algorithm. The main idea behind this algorithm is to search a many potential solutions in a short amount of time. In this algorithm, a set of solutions is initialized as the "population" [3]. Then, the population is evaluated using a scoring function.

Conference'17, July 2017, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Afterwards, a new population is created along with probabilities for selection, mutation, and crossover to modify the solutions included in the new population [3]. The time complexity for this algorithm is $O(\text{Size of population} * n)$. In general, this algorithm can achieve near optimal solutions in a short amount of time [3].

Overall, previous work can be used to conclude that there are many feasible ways to approach the 0-1 knapsack problem. As such, our team seeks to compare our findings to these studies to determine the feasibility of our own algorithms to solve the 0-1 knapsack problem.

4 ALGORITHMS

4.1 Approximation

For the approximation algorithm, we decided to use a greedy method. This algorithm first calculates the ratio of value to weight for each item and stores it as a list, requiring $O(n)$ time and space complexity. After, the list is sorted based on ratios in descending order, requiring $O(nlog(n))$ time complexity. Then, the algorithm iterates over the list sequentially, selecting the item with the highest ratio that does not exceed the weight threshold if added. The last step requires $O(n)$ time complexity. One addition made to this algorithm was to take the maximum of either the greedy solution or the item with the largest value in the knapsack. The pseudocode for this algorithm is given below:

Algorithm 1 Approximation

```
procedure APPROXIMATION
    ratios = [$\frac{v_i}{w_i}$] for $i$ in items
    sort(ratios) in descending order
    solution = []
    current_weight = 0
    current_value = 0

    for $i \leftarrow 1$ to $n$ {
        if $w_i + current\_weight < W$ {
            current_value += $v_i$
            current_weight += $w_i$
            append $i$ to solution
        }
    }

    if max($V$) > current_value {
        current_value = max($V$)
        solution = [argmax($V$)]
    }
    return current_value, solution
end procedure
```

Overall, the largest time complexity for this algorithm is $O(nlog(n))$ used for sorting. Therefore, the total time complexity for this algorithm is $O(nlog(n))$. Additionally, 3 arrays of size n are needed to store the item weights, item values, and item ratios. Therefore, space complexity for this algorithm is $O(n)$.

One of the biggest downsides to this algorithm is that it does not reach the optimal solution in many cases. This is because it does not consider combinations that include items that have already been seen. Additionally, the greedy algorithm does not care about the weight of the item added as long as its ratio is the highest at the current iteration and that the item can be added to the knapsack. However, this may result in a combination of future items that are not considered that would result in a better solution. Therefore, this algorithm is not guaranteed to find an optimal solution. However, the algorithm can find a near optimal solution in a short amount of time. Given that the time complexity for this algorithm is $O(nlog(n))$, it can be utilized on problems of large sizes quickly. Additionally, the solution that this algorithm obtains will usually be relatively close to the optimal solution. Therefore, it provides a way to get a reasonable solution in a short amount of time.

While this algorithm will get a reasonable solution in most cases, it is not always guaranteed to obtain a reasonable solution. The original greedy algorithm without taking the maximum of the greedy solution or the maximum valued item performs arbitrarily bad [1]. This means that the algorithm is not guaranteed to be close to the optimal solution at all. However, the modification to included maximization guarantees that the solution will be at least half the optimal solution [1]. Essentially, the greedy algorithm with a maximization of either the found solution or the largest valued item is a 2-approximation algorithm.

4.2 Branch and Bound

Applying the Branch and Bound algorithm to knapsack, requires us to define an appropriate upper bound for a partial solution. If the upper bound of a partial solution is smaller than the current best, then it is possible to prune the current node and its sub-problems. Like backtracking, a node is a state that defines a partial solution. We define the level of a node as the number of items that are considered to put into a state. For example, if the problem has ten items, a node of level five has as partial solution that where it is possible to have some combination of the first five items. The Node will also have to remember its current total weight and current total profit as well as the items chosen for it to reach that state. Therefore, we will define a Node as an object that has level, current weight and profit, and its item state. Items are assumed to be presorted by ratio of profit to weight from greatest to least. This way, an upper bound can be easily calculated from the greedy solution.

The Branch and Bound algorithm has a similar theoretical worse case as backtracking but with an added linear term due to $O(n)$ worse case for greedy bound calculation at $O(n * 2^n)$ where all 2^n states are examined. Branch and Bound will tend towards the behavior of backtracking occur if every element has the same profit to weight ratio and the number of items is sufficiently large compared to the total allowed weight. In this example, the calculated upper bound will always be the same so long as there are items left for fractional knapsack to pick. Since the profit to weight ratio is the same, the overall profit will be the same too. Since the upper bound profit is the same, the maximum profit cannot exceed the upper bound and will almost never be pruned by Branch and Bound's conditions.

Branch and Bound is quite slow compared to all the other algorithm because its worse case has to consider all possible states. However, if we set the initial upper bound as the solution of the approximation, we can ensure that the returned solution will be at least as good as the approximation with more potential pruning. The upside to Branch and Bound is that it will always find the optimal solution if given enough time. Given the above configure of a preset upper bound, it will also always return a configuration better than the approximation even if the algorithm is not allowed to completely finish.

Algorithm 2 Branch and Bound

```
procedure UpperBound(Node, Items, Weight)
    if Node.weight > Weight {
        return 0
    }
    sub_q_items = Items[Node.level+1]
    sub_q_weight = Weight - Node.weight
    greedysol = Greedy(sub_q_items, sub_q_weight)
    return Node.weight + greedysol
end procedure
procedure condInsert(pq, Node, Weight, Items)
    ubound = UpperBound(Node, Items, Weight)
    if Node.weight > Weight or ubound < maxProfit{
        return
    }
    if Node.profit > Weight{
        maxState, maxProfit = Node.state, Node.profit
    }
    pq.put(Node)
end procedure

procedure Branch and Bound(Items, Weight)
    curr = Node(-1, 0, 0, [])
    pq = PriorityQueue(key=node.profit)
    pq.put(curr)
    Global maxState, maxProfit = Approx(curr, Items, Weight)
    while not pq.empty() {
        currState = pq.get()
        Taken, NotTaken = expand(currState)
        condInsert(Taken)
        condInsert(NoteTaken)
    }
    return maxProfit, maxState
end procedure
```

4.3 Simulated Annealing

Our simulated annealing algorithm breaks down the knapsack problem to find the best solution. Our algorithm first creates an initial random solution. From here, the algorithm begins the simulated annealing process. This process involves starting with a high initial temperature to determine randomness. This temperature is slowly decremented as we modify our initial random solution by flipping the inclusion of different items in the knapsack. As we find better solutions, we keep track of those as our current solution and iterate until our temperature reaches our stopping temperature, which serves as the terminating point. Once the stopping temperature is reached, the best found solution is returned by the algorithm.

The simulated annealing algorithm does not guarantee we arrive at the optimal solution, but as a local search, it tends to find strong approximate solutions. In our case, our algorithm performs fairly well for the knapsack problem.

The time complexity of our simulated annealing algorithm is based on the starting temperature as well as the rate at which we cool the starting temperature until it reaches the final stopping temperature. In our algorithm, we choose the starting temperature as $100 + len(values) * 5$, where values are the values for each item. Our cooling rate is represented by $1/(10+0.1*len(values))$. We also choose our stopping temperature to be 0. Therefore, the time complexity will be our $startingtemperature/coolingrate$. This gives us a time complexity of $O(n^2)$, where n is the length of our values. The space complexity of our simulated annealing algorithm is $O(n)$, where n is the length of our values. This is because we perform the knapsack problem where we modify a solution of size n and keep the best solution. Therefore, the space complexity must be $O(n)$ with our auxiliary data structure to keep our solution.

There are several strengths and weaknesses with simulated annealing. With regards to strengths, simulated annealing can escape local optima through uphill moves. In addition, the algorithm is able to handle complicated search spaces and objective functions. With regards to weaknesses, it is difficult to find the optimal parameters to tune the algorithm based on varying data sets. The final solution depends heavily on the chosen parameters for the algorithm. In our simulated annealing algorithm, we used automated tuning based on the size of our datasets which allowed us to gain better results. Our decisions for approaching this algorithm were largely based on the ideas taught in lecture to design components.
In the pseudocode below, keep track of the weights and values of solutions as they are being generated. The pseudocode assumes that these weights and values are being tracked along with current solutions.

Algorithm 3 Simulated Annealing

```
procedure KnapsackValue(Knapsack, Weights, Values)
    totalWeight = value of Knapsack solution's item weights
    totalValue = value of Knapsack solution's item values
    return totalWeight, totalValue
end procedure
procedure InitializeSolution(weights, capacity)
    currentSolution = $[False] * len(weights)$
    currentWeight = 0
    randomize order of items
    for item in random order:
        if $currentWeight + itemWeight \leq capacity$
            currentSolution[item] = True
            currentWeight = currentWeight + itemWeight
    return currentSolution
end procedure

procedure SimulatedAnnealing(values, weights, capacity, stoppingTemp, cutoff, seed)
    currentSolution = InitializeSolution(weights, capacity)
    startTemp = $100 + len(values) * 5$
    coolRate = $1/(10 + 0.1 * len(values))$
    while startTemp > stoppingTemp:
        neighborSol = flip random item in currentSolution
        if neighborSolWeight ≤ capacity:
            delta = neighborSolValue − currentSolutionValue
            if delta > 0 or random.random() < math.exp(delta / startTemp):
                currentSolution = neighborSol
    return currentSolutionValue, currentSolution
end procedure
```

4.4 Hill Climbing

The hill climbing algorithm is a local search algorithm designed to converge to local maximums across the solution space with the hope of discovering the global maximum (the optimal solution. First, the evaluation function returns the total value of the items picked in the solution. If the total weight of the items picked exceeds the total capacity of the knapsack, then the evaluation returned is the negative of the total weight instead, as to penalize the solution more for being more overfilled. The algorithm then iterates until the cutoff time is reached. On each iteration, it randomly restarts at a solution (which is an array comprised of 0s and 1s) representing which items to pick. The starting solution is always purposefully sparse, as the algorithm struggled to remove items effectively much more than adding them. With 20% chance, the algorithm flips a bit of the solution at random. Otherwise, it iterates through the solutions neighborhood, which is comprised of every solution that's just one bit flip away. It then finds the neighbor with the largest evaluation, and if that best neighbor is better than the current solution, then the solution is replaced. If the solution has no better neighbors, then the solution is a local max or a peak, which means that the iteration is over. At that point, the peak is compared to the all-time best solution found, and the better of the two is chosen.
All of this can be seen in the following pseudo-code:

Algorithm 4 Random Restart Hill Climbing

```
procedure Eval(solution, weights, values, capacity)
    totalWeight = sum(elementMult(solution, weights))
    totalValue = sum(elementMult(solution, values))
    if weight > capacity
        totalValue = - totalWeight
    endif
    return totalValue
end procedure

procedure HillClimbing(values, weights, capacity, cutoff, seed)
    setSeed(seed)
    n = len(values)
    bestSolution = array of 0s with length n
    bestVal = 0
    startTime = currTime()
    while currTime() - startTime < cutoff
        sol = array of length n with entries from set {0, 1} with probability dist {0.99, 0.01}
        solVal = Eval(sol, weights, values)
        peakFound = False
        while not peakFound
            if random number in range [0, 1) < 0.2
                sol = flip random bit in sol
                solVal = Eval(sol, weights, values, capacity)
            else
                bestNeighborVal = -inf
                bestNeighbor = empty array
                for i < n
                    neighbor = flip bit i in sol
                    neighborVal = Eval(neighbor, weights, values, capacity)
                    if neighborVal > bestNeighborVal
                        bestNeighborVal = neighborVal
                        bestNeighbor = neighbor
                    endif
                endfor
                if bestNeighborVal > solVal
                    solVal = bestNeighborVal
                    sol = bestNeighbor
                else
                    peakFound = True
                endif
            endif
        endwhile
        if solVal > bestVal
            bestSolution = sol
            bestVal = solVal
        endif
    endwhile
    return bestVal, bestSolution
end procedure
```

Hill climbing is a good algorithm to approximate a solution, but isn't guaranteed to generate the optimal solution, even with unlimited time as the algorithm is stochastic. Hill climbing can also get stuck at local maxes pretty easily, since it only climbs upward from any given solution.

To further improve the likelihood of hill climbing reaching the optimal (or close) solution, I implemented random restarts which allows the algorithm to explore different starting points until a cutoff time is reached. This will allow it to reach different local maxes and pick the highest one out of the ones it reaches. To stop hill climbing from getting stuck at the same local maxes, I also implemented a randomized bit flip that happens 20% of the time, instead of choosing the best neighbor. This will allow the algorithm to deviate from certain paths it would otherwise take and potentially get stuck on. Both of these decisions made were inspired by the lecture slides which mentioned these solutions for the problems we found with the basic hill climbing algorithm.

A final modification was made to the random restart code, which made it so that new solutions generated were very sparse with choosing items. 99% of the time, an item would not be chosen, and this number was determined empirically, as the algorithm performed better at adding items that were missing from the solution than subtracting items that were unnecessary.

5 EMPIRICAL EVALUATION

Platform description:
Language - Python
Computer specifications - defined in each subsection

For our experimental procedure, we ran our local search algorithms in Python multiple times on each trace varying the seed and the cutoff time. We decided to perform our experiments this way to generate enough data for plots and to determine runtime complexity and solution quality for various parameters. For our branch-and-bound and greedy approximation, we did not utilize a seed and only ran the traces once. We decided to not run multiple instances because both these algorithms are deterministic, so the results should be the same regardless of running multiple instances.

The main evaluation metric used for each algorithm was relative error, which can be used to determine the quality of a solution. Relative error for each algorithm is calculated as follows:

$$RelativeError = \frac{OptimalSolution - AlgSolution}{OptimalSolution} \tag{3}$$

In addition to relative error, runtime complexity was used as an evaluation metric for each algorithm to determine the feasibility of coming up with an optimal solution in a reasonable time. Using these metrics, we established different evaluation criteria for our algorithms. For our branch-and-bound algorithm, our evaluation criteria was the expected runtime. This is because the branch-and-bound algorithm is expected to reach the optimal solution if given an unlimited amount of time. Therefore, it is more important to measure the runtime complexity instead of solution quality. For the approximation algorithm, our evaluation criteria was the solution quality and runtime complexity. The approximation algorithm is not expected to find optimal solutions often, but it is expected to run quickly. Therefore, we can evaluate the algorithm based on both these metrics. Additionally, we can use these metrics to develop approximation guarantees for the approximation algorithm. The evaluation criteria for our local search algorithms was the probability of finding an optimal solution within specified time constraints. To measure this, we utilized a qualified runtime distribution (QRTD) and solution quality distribution (SQD). The QRTD uses a fixed solution quality and varies the time to determine the probability of reaching the specified solution quality after running the algorithm for various amounts of time. The SQD measures the probability of reaching various solution qualities within a fixed amount of time. Both plots were created using trace files generated by our local search algorithms for the datasets large_1 and large_3. The plots utilized 50 traces from each algorithm and dataset, totalling 200 traces to generate the plots. In addition to these plots, a boxplot was also used to evaluate the runtimes of our local search algorithms. We explain our evaluation of each individual algorithm in more detail below.

5.1 Approximation

	Approximation		
Dataset	Time (s)	Total Value	RelError
small_1	0.00095	294.0	0.0059
small_2	0.0020	1018.0	0.0059
small_3	0.0020	35.0	0.00
small_4	0.0010	16.0	0.44
small_5	0.0020	481.0694	0.00
small_6	0.0010	52.0	0.00
small_7	0.0020	102.0	0.049
small_8	0.0025	9751.0	0.0016
small_9	0.00097	130.0	0.00
small_10	0.0025	1019.0	0.0059
large_1	0.0039	8817.0	0.037
large_2	0.0085	11227.0	0.00098
large_3	0.018	28834.0	0.00080
large_4	0.048	54386.0	0.0022
large_5	0.073	110547.0	0.000706
large_6	0.20	276457.0	0.00028
large_7	0.32	563605.0	0.000075
large_8	0.0045	1487.0	0.018
large_9	0.0076	1604.0	0.019
large_10	0.018	4552.0	0.0031
large_11	0.031	9046.0	0.00066
large_12	0.069	18038.0	0.000721
large_13	0.17	44350.0	0.00014
large_14	0.33	90199.0	0.000055
large_15	0.0040	2375.0	0.0093
large_16	0.0065	2649.0	0.018
large_17	0.018	7098.0	0.0027
large_18	0.046	14374.0	0.0011
large_19	0.069	28827.0	0.0032
large_20	0.17	72446.0	0.00081
large_21	0.41	146888.0	0.00021

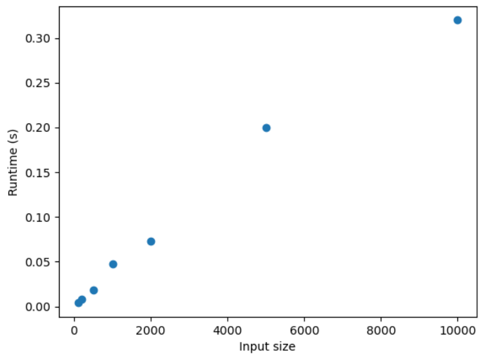
The approximation algorithm was run on all instances on Windows. The machine used an Intel 8-core CPU as well as 16 Gigabytes of RAM. The cutoff time was set at 600 seconds, but was never exceeded since all runtimes were less than 1 second.

Looking at the table above, it is clear that the approximation algorithm runs quickly and often finds a near-optimal solution. However, there are some edge cases in which it performs poorly, such as for the dataset small_4. Despite this one poor solution, it is evident that the approximation algorithm is within its approximation bounds detailed in the algorithm section. Since our highest relative error is 0.44, then we find that the algorithm solution achieves a solution that is 0.66 the amount of the optimal solution. Since the greedy approach we used is a 2-approximation algorithm, we can use the following inequality to compare the approximation solution and optimal solution:

$$\frac{OPT}{2} \leq A \leq OPT \tag{4}$$

Since 0.66 is our lowest bound on the values of the approximation solution compared to the optimal solution, we find that the lowest bound for our experiments are within the approximation guarantees for our approximation algorithm.

Another important aspect of our approximation algorithm is its runtime complexity. Theoretically, the runtime complexity for our algorithm is $O(nlog(n))$. To evaluate our emperical runtime complexity in comparison, we created a graph using the datasets from large 1-7.



As can be seen in the graph, our plot of runtimes for approximation is relatively linear. Therefore, our emperical runtime is upper bounded by $O(nlog(n))$ and matches the expected runtime complexity.

5.2 Branch and Bound

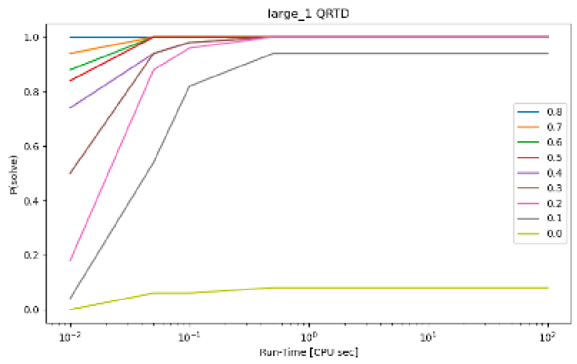
	Branch and Bound		
Dataset	Time (s)	Total Value	RelError
small_1	0.00041	295.0	0.00
small_2	0.00059	1024.0	0.00
small_3	0.00019	35.0	0.00
small_4	0.0003	23.0	0.00
small_5	0.00031	481.0694	0.00
small_6	0.00099	52.0	0.00
small_7	0.00029	107.0	0.00
small_8	14.017	9767.0	0.00
small_9	0.00021	130.0	0.00
small_10	0.001	1025.0	0.00
large_1	0.0027	9147.0	0.00
large_2	0.0057	11238.0	0.00
large_3	0.0092	28857.0	0.00
large_4	0.011	54503.0	0.00
large_5	0.12	110625.0	0.00
large_6	0.45	276457.0	0.00
large_7	0.39	563647.0	0.00
large_8	0.015	1514.0	0.00
large_9	0.015	1634.0	0.00
large_10	0.015	4566.0	0.00
large_11	0.033	9052.0	0.00
large_12	0.21	18051.0	0.00
large_13	0.29	44356.0	0.00
large_14	2.45	90204.0	0.00
large_15	0.0054	2397.0	0.00
large_16	0.55	2697.0	0.00
large_17	0.96	7117.0	0.00
large_18	58.88	14390.0	0.00
large_19	3600	28919.0	0.00
large_20	3600	72505.0	0.00
large_21	3600	146919.0	0.00

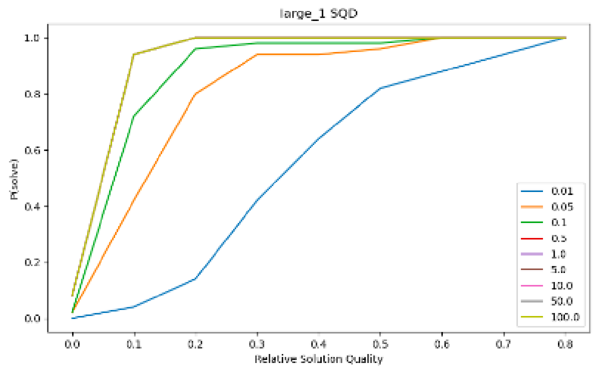
The Branch and Bound algorithm was run on a Linux computing cluster hosted in the coda building. This machine has an Intel sever level CPU with 48 cores and 32 Gigabytes of RAM. Branch and bound was ran on all instances with a cutoff time of one hour and was able to find the optimum solution for every dataset instance. However, it is notable that for instance large_19, large_20, and large_21, the algorithm finds the best solution but never finishes. In fact, in testing, increasing the cutoff time to 5 or 6 hours still doesn't allow the algorithm to terminate for those instances.

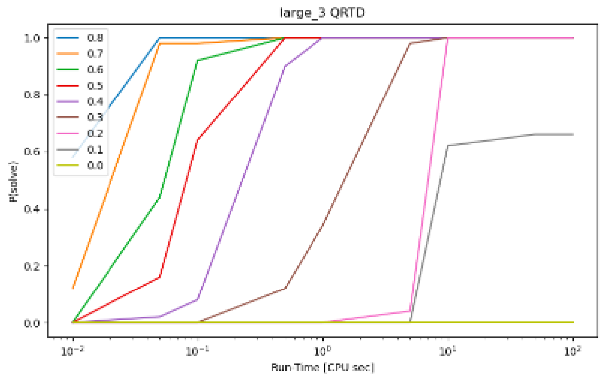
5.3 Simulated Annealing

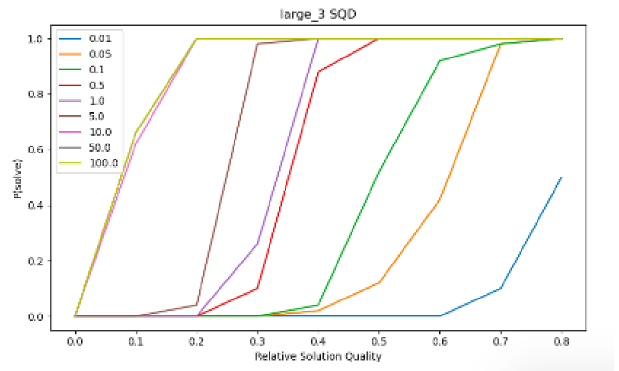
	Simulated Annealing		
Dataset	Time (s)	Total Value	RelError
small_1	0.0014	295.0	0.00
small_2	0.33	1016.0	0.0078
small_3	0.00026	35.0	0.00
small_4	0.00026	23.0	0.00
small_5	0.15	463.5701	0.036
small_6	0.0076	52.0	0.00
small_7	0.00034	107.0	0.00
small_8	0.00055	9688.0	0.0081
small_9	0.00	130.0	0.00
small_10	0.37	1019.0	0.0059
large_1	1.60	8940.0	0.023
large_2	8.25	10631.0	0.054
large_3	79.055	26950.0	0.066
large_4	460.60	52217.5	0.042
large_5	495.41	64629.0	0.42
large_6	524.13	143352.0	0.48
large_7	562.75	276318.6	0.51
large_8	2.066	1265.0	0.16
large_9	3.23	1440.0	0.12
large_10	84.19	3934.0	0.14
large_11	453.82	7160.0	0.21
large_12	407.70	12696.3	0.30
large_13	261.37	29069.0	0.34
large_14	397.53	56252.0	0.38
large_15	1.83	2197.0	0.083
large_16	2.95	2495.0	0.075
large_17	82.00	6112.0	0.14
large_18	398.19	11650.0	0.19
large_19	62.59	22006.0	0.24
large_20	468.65	52797.0	0.27
large_21	560.12	103612.8	0.29

The simulated annealing algorithm was run on all instances for each of 10 seeds (1-10) on MacOS. The machine used has an Intel 8-core CPU as well as 16 Gigabytes of RAM. The max runtime was set as a cutoff at 600 seconds with the results in the data table being the aggregate of the averages of each of the 10 seeds for each instance. We can see that for larger datasets, a local search such as simulated annealing began to see a drop in performance relative to the RelError.
Below, you can find the QRTDs and SQDs of simulated annealing for the instances large_1 and large_3 across 50 different seeds:

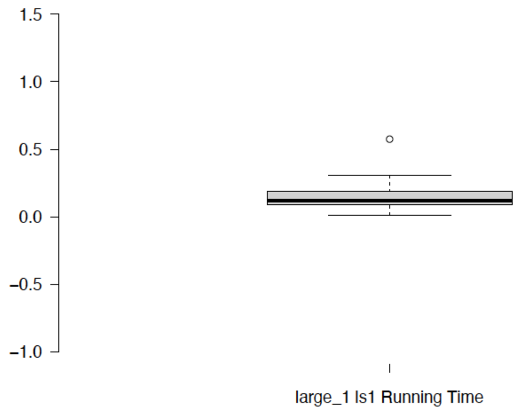


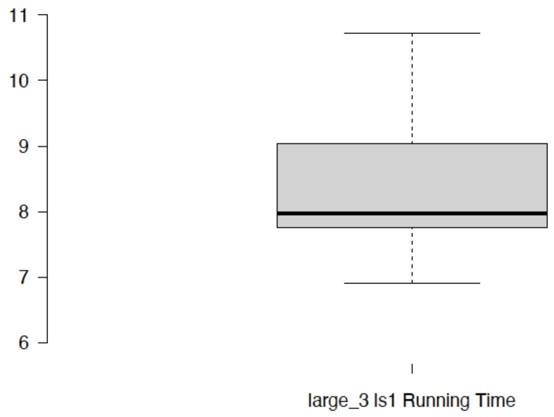






These charts all demonstrate that our simulated annealing algorithm has very standard results. We can see that results are consistent between the two different datasets. As time increases, the likelihood our algorithm solves the problem increases at a strong rate.





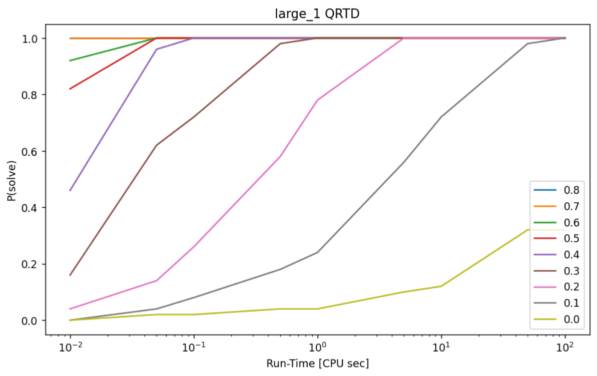
The box plots above demonstrate the distribution of running times for 50 executions of the simulated annealing algorithm on each data set. We can see that for a smaller dataset, the run time is more consistent, while for a larger dataset, there is more variance among the different running times.

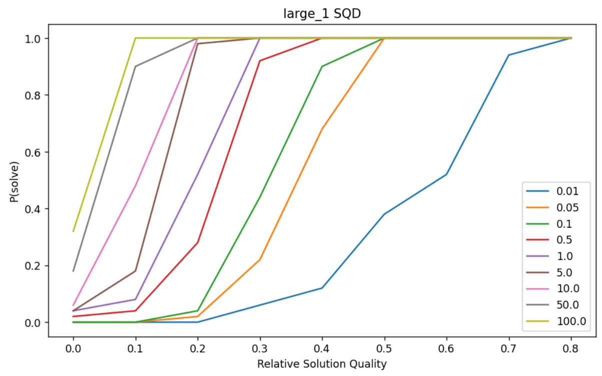
5.4 Hill Climbing

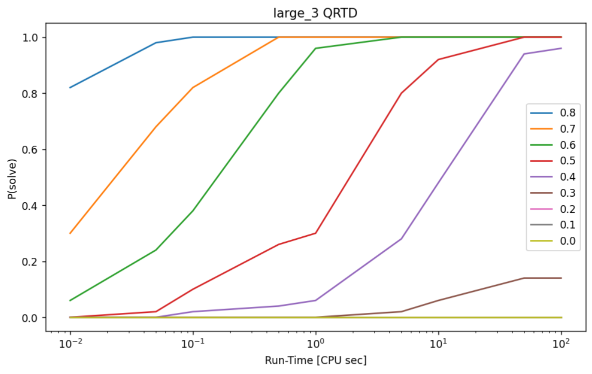
	Hill Climbing		
Dataset	Avg Time (s)	Avg Value	Avg RelError
small_1	0.0096	295.0	0.00
small_2	0.0097	1024.0	0.00
small_3	0.0035	35.0	0.00
small_4	0.00067	23.0	0.00
small_5	0.016	481.0694	0.00
small_6	0.0050	52.0	0.00
small_7	0.00085	107.0	0.00
small_8	0.066	9767.0	0.00
small_9	0.00097	130.0	0.00
small_10	0.0086	1025.0	0.00
large_1	219.067	9147.0	0.00
large_2	45.036	10555.0	0.061
large_3	401.18	18590.0	0.36
large_4	228.61	23504.0	0.57
large_5	504.81	33742.5	0.70
large_6	77.85	46330.0	0.83
large_7	50.88	74568.0	0.87
large_8	56.72	1433.0	0.054
large_9	239.31	1416.0	0.13
large_10	280.67	3354.0	0.27
large_11	422.62	6166.0	0.32
large_12	189.31	11324.0	0.37
large_13	148.48	26743.0	0.40
large_14	270.029	52187.6	0.42
large_15	15.16	1896.0	0.21
large_16	99.60	1896.0	0.30
large_17	74.46	3817.0	0.46
large_18	230.70	6990.0	0.51
large_19	210.97	13019.0	0.55
large_20	33.36	31205.0	0.57
large_21	301.32	61119.0	0.58

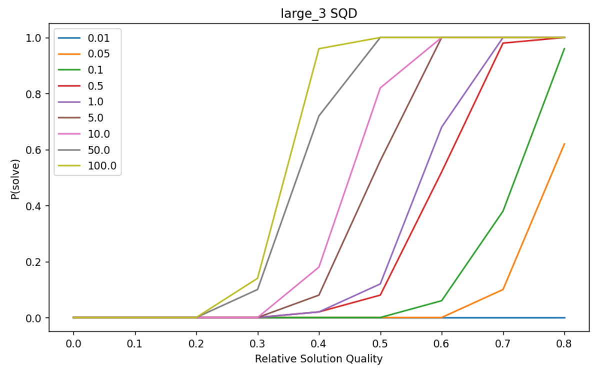
This algorithm was run in parallel across all instances for each of 10 seeds (1-10) on a Linux computing cluster hosted in the coda building. This machine has an Intel sever level CPU with 48 cores and 32 Gigabytes of RAM. Each runtime had a 600 second cutoff, and the averages of the last timestamps and values were taken for each instance across all 10 seeds.

Below, you can find the QRTDs and SQDs of hill climbing for the instances large_1 and large_3 across 50 different seeds:



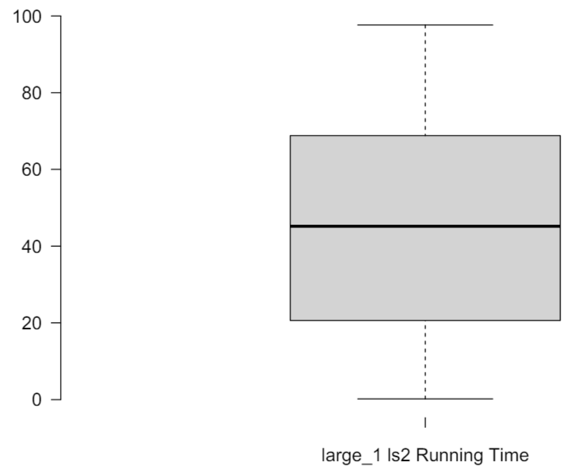


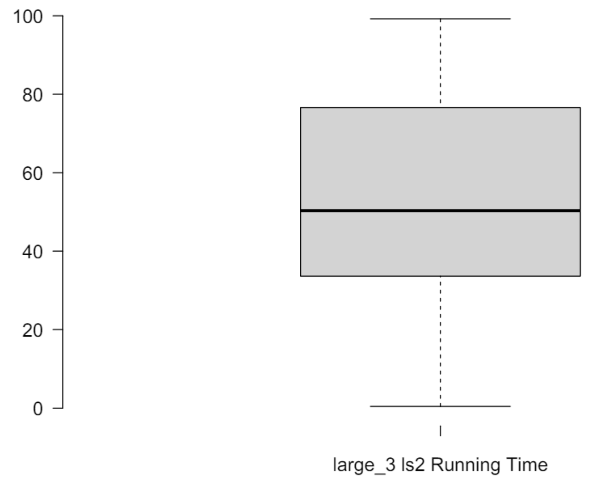




These charts all paint a very similar picture for hill climbing, as within a short amount of time, the QRTDs show that certain relative errors are attainable (higher relative errors for the smaller instance of large_1 than large_3). As the amount of time increases, the likelihood of different runtimes solving up to a certain quality seems to increase across the board. The SQDs paint a similar picture, as the highest runtime lines always have the highest solution rates for any relative errors, even across different instances.

It's also interesting to note how consistent the results are across both instances, with the QRTD for large_3 effectively being the same as for large_1 just shifted down, and the SQD for large_3 being a right-shifted version of the large_1 SQD.





The box plots above show the running times it took for hill climbing to find the best solutions it could in datasets large_1 and large_3. These are the same runs that generated the QRTD and SQD diagrams earlier. These plots show that hill climbing pretty consistently will use whatever time is allotted for it, but takes significantly more time to find the next best solution. While the runs lasted 100 seconds, both datasets had median runtimes for the best run around 50 seconds, which meant that the last 50 seconds of at least half the runs were not able to generate a better solution. Also, the median and 1st and 3rd quartile runtimes for large_3 are slightly higher than large_1, which supports the observations that hill climbing struggles more with larger datasets.

6 DISCUSSION

In general, we found that the branch and bound algorithm performs the best in terms of solution quality. Our branch and bound algorithm achieved the optimal solution in every case, which is not the case for our other algorithms. However, the time complexity of branch and bound makes it incredibly slow as the data size increases. Therefore, we consider our local search and approximation algorithms to find an optimization of runtime and solution quality. In terms of solution quality, both local search algorithms perform better than the approximation algorithm since they achieve more optimal solutions with a short runtime. However, our approximation algorithm performed better on the large datasets compared to both local search algorithms in terms of solution quality and runtime. This is likely due to the randomization of both local search algorithms slowing down the convergence to an optimal solution as data size increases. Comparing simulated annealing and hill climbing, we found that hill climbing performs slightly worse on average because the relative error was higher for most instances despite using the same cutoff times. Finally, in terms of runtime our approximation performed the best with a runtime of less than one second for each instance. Overall, we found that local search algorithms and branch and bound are strong methods for solving the 0-1 knapsack problem with small datasets. However, as the data size increases, it is important to consider whether solution quality or time complexity is more important when selecting an algorithm to solve the problem.

6.1 Approximation

Based on our results, the approximation algorithms performs as expected based on our evaluation criteria of solution quality and runtime complexity. For all instances, the algorithm completes quickly following the expected $O(nlog(n))$ runtime complexity. Additionally, the quality of solutions follow the expected approximation ratio of being at least half the optimal solution. From our results, we find that the approximation algorithm reaches the optimal solution for four of the experiments. Additionally, each of those experiments is from the small datasets with the largest of them having 20 items. This makes sense as introducing more items increases the chances that there is a combination of items that are optimal and not considered by the greedy approximation algorithm.

This can be seen in our results as our approximation algorithm never finds the optimal solution for the large datasets. It is worth noting that the quality of the worst solution occurs for the small_4 dataset but still falls within range of the approximation ratio. Consider the relative error for small_4. For this dataset, the algorithm achieved a relative error of 0.44, which is quite poor. This makes sense when looking into the data for small_4. The data is given as [(6,2), (10,4), (12,6), (13,7)] where the first number is the item value and the second is the weight. Additionally, the capacity for this problem is 11. As such, the knapsack can only hold 2 of these items regardless of which items are chosen. The algorithm selects the first and second items based on ratio, but the optimal solution picks the second and fourth item. Since the data for small_4 involves small values, any deviation from the optimal solution will have a greater impact on solution quality. Additionally, the algorithm is presented with a case where it should skip the item with the highest ratio. By picking the items with the highest ratio, the weight in the knapsack is 6 and no other items can be added due to their weight being greater than 5. However, the knapsack can be filled to 11 using the second and fourth item. As such, picking the first two items leaves the knapsack in a state where its space is under utilized. Therefore, the algorithm is expected to have poor solution quality for this dataset.

Despite there only being one case in which the approximation algorithm performs poorly for these datasets, it should be noted that the approximation algorithm is not guaranteed to perform this well. Given completely different datasets, this algorithm cannot be expected to perform as well. The only expectation that can be made about the approximation algorithm regardless of datasets is that it will achieve a solution that is at least half the optimal solution.

6.2 Branch and Bound

The Branch and Bound Algorithm performs quite well and generally finishes under 3 seconds for most instances. There are five exceptions small_8 and large_18 to large_21. On closer examination of small_8, this instance is very close to the worse case scenario of Branch and bound. The ratios for all items is very close to one and the total weight is chosen such that around half the items must be chosen to get close to the optimum. The close ratios ensure that the upper bound does not prune a state often and the chose weight ensures that around half the items must be chosen before we start reaching dead-ends. The two combination makes sure that even though small_8 only has 23 items, it still runs for a relatively long time. The problematic larger datasets also exploit the two weakness of Branch and Bound described above so has a long run time. It is worth noting that large_18 still finish in under a minute but a further doubling in size from large_18 to large_19 exemplifies the exponential nature of the algorithm where the algorithm does not finish running even when the cutoff is expanded to 5-6 hours. Branch and Bound does find optimum solution for all instances, however, a tighter bound or further optimization may be needed to tackle large_18 to large_21 since the defined upper bound is evidently not effective enough in pruning the node states. It is worth noting that the effectiveness of pruning is still evident even for those problematic instances. In tests without proper bounds in problematic instances, the RAM usage rose very quickly. When usage reached 15 Gigabytes, the test was ended. However, when proper bounds were introduced, the pruning ensures that the RAM usage stabilizes at less than 1 Gigabyte.

In future works, it is worth noting that bound calculation is a large part of Branch and Bound. Therefore, an improvement in time complexity in bounds calculation may speed up Branch and Bound significantly. This can be done by pre-calculating a running sum of weights and profit of the sorted items array, then using binary search to find the largest cumulative weight that is still smaller than the maximum allowable weight.

6.3 Simulated Annealing

Our simulated annealing algorithm performs well given that it is a local search when applied to the knapsack problem. It performs as expected for the algorithm.

It's solution is always worse or equal to the Branch and Bound algorithm. As datasets get larger, there is also a drop off in performance which can be seen in the increasing RelError in our empirical evaluation. For all instances, the algorithm completes within the expected time complexity and takes only the necessary space complexities mentioned in the algorithm section of this paper. For smaller datasets, the simulated annealing algorithm is able to quickly come to a solution with only minor RelError when compared to the true solution. When compared to our other local search, hill climbing, there is a noticeable improvement in run time for smaller datasets for the simulated annealing algorithm. As datasets get larger, hill climbing is able to arrive at a solution sooner, but the average RelError performance is worse.

It is likely that larger datasets result in the performance issues mentioned in the algorithm section, where the hyperparameters are not tuned efficiently, resulting in worse solution search spaces. In future works, it may be useful to explore comparing different hyperparameter results to determine optimal settings in order to reduce RelError.

6.4 Hill Climbing

Within relatively short time frames, the hill climbing algorithm performs decently well, especially for smaller datasets as it's able to achieve extremely low or 0 relative errors within a second. As the datasets get larger, the hill climbing algorithm has a lot more to explore, and that's where its inefficiencies become more apparent. having high relative errors on instances such as large_7 and large_21 makes sense, since traversing through the neighborhood of any given solution takes $O(n)$ time. Thus, going through a new neighborhood will take more and more time that could otherwise be used to find better solutions.

In general, this version of the hill climbing algorithm does seem to generate decent outputs, and given enough time, it would seem to converge towards the optimal value. However, that required time grows very quickly with the size of the instance, and due to the inefficient nature of hill climbing being able to reevaluate solutions it had before and get stuck at the same local optima, this algorithm in its current state is not the best way to estimate the knapsack problem.

7 CONCLUSION

In summary of our findings, our research found that branch and bound is best for solving the 0-1 knapsack problem optimally. While our other algorithms can solve problems almost optimally in a quicker runtime, it does not guarantee that the quality of the solution is great. The greedy approximation algorithm performs well for this dataset, but is not guaranteed to always perform as well as seen in our experiments. As such, it should be used alongside other algorithms, such as local search algorithms to gain a baseline solution. Finally, we conclude that our local search algorithms can be incredibly volatile with the quality of their solutions. In future research, it would be important to look at other local search algorithms, such as genetic algorithms, or to modify the neighborhood selection and initialization of our local search algorithms.

REFERENCES

[1] Chandra Chekuri. 2009. *Approximation Algorithms*. https://courses.engr.illinois.edu/cs598csc/sp2009/Lectures/lecture_4.pdf.

[2] Salem Hildebrandt and Christopher Hanson. 2016. *0-1 Knapsack Optimization with Branch-and-Bound Algorithm*. https://www.micsymposium.org/mics2016/Papers/MICS_2016_paper_42.pdf.

[3] Maya Hristakeva and Dipti Shrestha. 2005. *Different Approaches to Solve the 0/1 Knapsack Problem*. https://micsymposium.org/mics_2005/papers/paper102.pdf.

[4] Xiaohui Pan and Tao Zhang. 2018. Comparison and analysis of algorithms for the 0/1 knapsack problem. *Journal of Physics: Conference Series*, 1069, 1, (Aug. 2018), 012024. DOI: 10.1088/1742-6596/1069/1/012024.