Max Barshay
March 11, 2020

Knapsack Final Report

1) **Introduction**

In this report I will compare and contrast four different algorithms for solving the 0-1 knapsack problem. The approaches I will explore are exhaustive enumeration, greedy, dynamic programming and branch and bound. Each approach will be in its own section. In addition, I will discuss the data structures I used, the complexity of each approach as well as the pros and cons of each approach. Finally, I will briefly suggest possible improvements for each algorithm that still fit within their respective paradigms.

2) **Exhaustive Enumeration Approach**

One way to solve the 0-1 knapsack problem is by trying every combination of items. In order to implement this approach, I had to generate all possible subsets of length n, where n is the total number of items under consideration. Luckily, having coded a gray code generator in a previous project, I realized that creating all gray codes of length n, does a lot of the work for creating all possible subsets of size n. I then created a simple method that converts all of the gray codes into lists of all the combination of indexes. This method simply looped over the characters of the gray code and if the character in the gray code was 1, I included the index in the subset, and didn't include the index otherwise. Using all the possible combinations of indices just generated,

I calculated the total value of each and every one and found the optimal solution which was the combination of indices that generated the highest total value, while still weighing less than the capacity.

Since I must check each combination of a set of length n, I know that I must check 2^n different combinations, where n is the total number of items that are under consideration. In order to test each potential combination, I had to loop over all 2^n possible combination, and within each combination in the worst-case loop over all n indices within the given combination. Thus, my algorithm is $O(n*2^n)$ in the worse case. Empirically, my algorithm took roughly .62 seconds on the Easy20 test data.

A pro of this algorithm is that it will clearly always find the correct answer assuming it is implemented correctly. Another pro is that it is not hard to think of nor implement. A major con of this algorithm is that it does not try solutions in any intelligent way. It is possible that only one item will fit in the knapsack, but we are given 100 items, the algorithm will not stop after testing the 100 single items, it will instead try all 2^100 combinations even if the majority do not fit.

To improve this algorithm we could possibly find a more efficient way to come up with the subsets, although the time it takes to generate the subsets is quite small compared to the time the rest of the algorithm takes.

3) **Greedy Approach**

In order to implement my greedy algorithm, I decided to make an Item object so that I could change the order of the items, while still keeping track of their respective indices for reporting. I decided to sort my items in non-decreasing order of value to price ratio, breaking ties arbitrarily. I decided to order my items in this way because I know that greedy algorithms tend to have the best performance if the first items that they select are more “promising”, in other words more likely to be a part of the optimal solution. I figured that items that had a larger value to weight ratio were more likely to be a part of the optimal solution.

In order to implement this greedy algorithm, we must sort the items in non-increasing order of value to weight ratio. This will run in O(n*logn) time where n is the number of items that must be sorted. The complexity of the single for loop used to actually find the greedy solution gets masked by the time it takes to sort, and we are left with an overall runtime of O(n*logn). In my test runs of the algorithm, I found that greedy completes in between .05 and .07 seconds for the shortest and longest runs respectively.

A pro of this algorithm is that it is easy to code, and its sorting criteria is easy to come up with. In order to code this algorithm, we simply have to sort the items in non-increasing order of value to weight ratio, and then loop over each item and check if it fits in the knapsack. If it fits, add it to the knapsack, if it does not fit, skip that item and move on to the next item. *There are potentially other ways to sort the items as long as the items that come first are in some sense “more promising” items to add. In this case, it is clear that items that have a higher ratio of value to weight are more “promising” and hence more likely to be included in the final solution. *A major con of this algorithm is that it does not always find the correct solution. It is not hard to imagine a situation where this greedy algorithm selects items that have large value to weight ratios, but those items were not included in the final solution and thus the final solution greedy arrives at is suboptimal.

One possible improvement to make is a minor adjustment to the selection criteria. We still would have the items sorted in non-increasing order of value to weight ratios, however once we got to an item that could not fit in the knapsack, instead of just going on to the next item, we would compare the total value of all the previous items in the knapsack to the value of the item that currently does not fit and take the larger of the two values. Then resume the algorithm until we run into the next item that does not fit and repeat this process. It can be shown that this new “greedy” algorithm is a 2-approximation of the optimal solution, whereas the greedy algorithm that I implemented above is not a finite-approximation.

4) **Dynamic Programming Approach**

Implementing my dynamic programming solution to this problem, was fairly simple in terms of data structures. In order to store solutions to each level of subproblem, I used a n + 1 x C + 1 matrix (2d array), where n is the number of items and C is the capacity. To solve this problem using dynamic programming, I looped over each possible entry of the table in a nested for loop and solved the subproblem corresponding to that spot in the table using my recurrence relation. Each entry of the table represents a problem with some prefix of the number of items in the original problem (including all of the items) and some capacity, an integer less or equal to the original capacity. Using some criteria specified in my recurrence relation, I select the larger value of two relevant subproblems as the solution to the current sub-problem I am looking at. The value in the lower right entry of the 2d array is the solution to the original problem with all the items and full capacity.

In order to solve the original problem, we have to fill in each entry in the matrix. Filling in each entry is a O(1) operation since we just have to access the matrix and possibly make a few simple calculations. Since we are looping over n + 1 rows and C + 1 columns the time complexity is O(nC). My implementation took .04 and .26 seconds for easy20 and hard200 respectively.

A pro of this algorithm is that it is fast and correct. This algorithm is faster than brute force significantly, and it always produces the correct answer. The proof of correctness can be done by induction on the number of items, using the recurrence relation to help with the inductive step. A major con of this algorithm is that the recurrence relation can be difficult to come up with, especially for someone with little experience in dynamic programming. Another con of this implementation is that it returns only the optimal value for a solution, and not the items in the solution. In order to determine which items were selected, one has to trace back over the matrix of solutions to determine which items were a part of the optimal solution.

5) **Branch and Bound Approach**

In order to implement branch and bound I made a Node object, where each node represents a partial solution to the knapsack problem that was given. Each node stores the weight and value of the partial solution that it represents. As well as the level in the state space tree that it is in. It also stores an upper bound on the value of any solution that is in the subtree rooted at that node, using a bounding function soon to be described. To enable me to keep track of the items taken, each node also stores a list of nodes that it has traveled through. In order to search the state space tree in an efficient manor, I used best-first search where best refers to a node with a larger value for the bounding function. Searching in this way allows me to expand nodes that are more likely to have an optimal solution in their subtrees and thereby potentially limit the amount of time I have to spend looking through the tree, since I can remove subtrees from contention to be searched if that subtree's root has an upper bound smaller than my current maximum value. The bounding function that I used was to add to whatever the current nodes value is, the value of all the items that can be greedily added to the knapsack (in non-increasing order of value to weight ratio), as well as the fraction of the last item that could not fit such that the final fraction makes the capacity 100%.

Branch and bound functions can be very difficult to analyze asymptotically, since they are very dependent on the specific data that was used as well as the bounding function and the search method. However, regardless of those choices, there exists a scenario where the best possible solution is the solution that corresponds to the node in the last level and furthest to the right of the state space tree, and due to the values of the other nodes, we do not explore that node until the very end of our best-first search. This shows that in the worst case our algorithm is $O(2^n)$ (if you run this algorithm to completion). From my runs of the algorithm I found that it can run from between .04 seconds for easy20 to up to 60 seconds for hard200, in which I stopped it at 60 seconds because it had yet to find a solution.

A pro of this algorithm is that it is very easy to stop mid-run and return the best solution that it has computed so far. In my implementation we compute the best value as we search even if we do not plan to stop the algorithm early. This contrasts to dynamic programming where one does not need to keep track of the best answer as they go. One major con of this algorithm is that it is hard to implement, even with a good understanding of branch and bound. Another con is that it is very data dependent and the time it takes to find a solution can vary greatly depending on where the solution is located in the state space tree.

There potentially are bounding functions that are tighter than the one that I used while also still being correct. I decided to use this bounding function since I knew that it was correct, and I did not want to risk trying to find a tighter one that could in some cases be incorrect, which would prevent my algorithm from guaranteeing an optimal solution if run to completion.

6) Table

	Easy20	Easy50	Hard50	Easy200	Hard200
Exhaustive Enumeration	Value 726 Weight 519 0.61698s	Not run	Not run	Not run	Not run
Greedy	Value 692 Weight 476 0.047186s	Value 1115 Weight 247 0.04909s	Value 16538 Weight 10038 0.04985s	Value 4090 Weight 2655 0.06673s	Value 136724 Weight 111924 0.07238s
Dynamic Programming	Value 726 Weight 519 0.04202s	Value 1123 Weight 250 0.04491s	Value 16610 Weight 10110 0.06958s	Value 4092 Weight 2658 0.08198s	Value 137448 Weight 112648 0.26374s
Branch and Bound	Value 726 Weight 519 0.04035s	Value 1123 Weight 250 0.04420s	Value 16610 Weight 10110 0.04647s	Value 4092 Weight 2656 0.06506s	Not run

7) Conclusion – Recommendations

In conclusion, I would recommend dynamic programming to my client to solve this problem since I do not have any specific knowledge of their implementation needs. Despite the difficulty of coming up with the recurrence relation, the dynamic programming solution has many benefits. One being that it always returns a correct answer, which is probably the most important consideration to make when deciding which algorithm is optimal. Furthermore, it is much easier to actually implement than branch and bound. Its implementation is also simpler, the main work done in the dynamic programming algorithm is done within one if-else block and a simple check of the maximum of two values.

If the client was trying to solve a very large instance of the problem, then I would recommend the branch and bound implementation since it is very easy to cut short and return the best answer that has been found up to that point. If the client had a very large instance of the problem and they did not care about accuracy but wanted the algorithm to run very fast, then I would recommend the augmented greedy that I described in the improvements of greedy section. I would recommend augmented greedy over my greedy algorithm because my greedy algorithm has the potential to return a solution that is larger than the optimal solution by a factor of an arbitrary large constant, whereas the augmented greedy solution is guaranteed to be a within a finite factor larger than the optimal solution. Furthermore, I would not recommend exhaustive enumeration, despite its correctness, in any situation as both dynamic programming and branch and bound can solve small instances of the knapsack problem in significantly less time than exhaustive enumeration can.