Parth Ray (pray)

Final Project Writeup

This report clearly and concisely explains four algorithms that can be used to solve the 0-1 Knapsack problem. The four algorithms that were utilized were Exhaustive Enumeration, a Greedy approach using the largest value-weight ratio, Dynamic Programming, and a Branch and Bound algorithm. Below is an analysis of each algorithm, that will explain the strengths and weaknesses of each technique, allowing for a quick and simple choice when given a certain knapsack problem set. This table below shows the results of various testcases:

	Easy20	Easy50	Hard50	Easy200	Hard200
Exhaustive Enumeration	Value: 726 Weight: 519 Runtime: 0.02181930 seconds	Not run	Not Run	Not run	Not Run
Greedy	Value: 692 Weight: 476 Runtime: 0.00010180 seconds	Value: 1115 Weight: 247 Runtime: 0.00012400 seconds	Value: 16538 Weight: 10038 Runtime: 0.00006070 seconds	Value: 4090 Weight: 2655 Runtime: 0.00018190 seconds	Value: 136724 Weight: 111924 Runtime: 0.00019010 seconds
Dynamic Programming	Value: 726 Weight: 519 Runtime: 0.00101970 seconds	Value: 1123 Weight: 250 Runtime: 0.00120430 seconds	Value: 16610 Weight: 10110 Runtime: 0.01127640 seconds	Value: 4092 Weight: 2658 Runtime: 0.00538370 seconds	Value: 137448 Weight: 112648 Runtime: 0.09929729 seconds
Branch and Bound	Value: 726 Weight: 519 Runtime: 0.01326850 seconds	Value: 1123 Weight: 250 Runtime: 1.82224262 seconds	Value: 16610 Weight: 10110 Runtime: 1.44647014 seconds	* Value: 480 Weight: 82 Runtime: 60.31206512 seconds ** OutOfMemory	* Value: 7526 Weight: 4226 Runtime: 60.41254807 seconds ** OutOfMemory

* Program interrupted before completion

** Reason if program fails

Exhaustive Enumeration:

Exhaustive Enumeration essentially tries all the permutations for the Knapsack problem and returns the maximum profit value. This algorithm utilizes recursion as it is much easier to implement this way. Since this method executes all permutations for a given problem size, the theoretical time complexity is O(2^n) where n is the number of items in the Knapsack problem. This method only ran for the easy20 testcase because this method is by far the most time consuming and can only support up to 1000 recursion calls. Overall, this method had the longest runtime for the easy20 testcase, making it the slowest of the four algorithms. This algorithm should only be used if the problem has less than 20 items and you do not have time to implement more complex algorithms.

Greedy:

The largest value to weight ratio greedy algorithm was implemented for this approach. The basic idea is that you take the item with the largest value to weight ratio until you reach the capacity. This algorithm has a theoretical time complexity of O(n), where n is the number of items in the Knapsack problem. Although, this method was the fastest out of all four of the algorithms, it is not guaranteed to return the optimal solution. It will always return a solution, but it may or may not be the optimal one. This method is the best that it can be. Therefore, this algorithm should be chosen if speed is the utmost priority and an approximate solution is acceptable.

Dynamic Programming:

The Dynamic Programming approach essentially creates a table that is of size n x C, where n is the number of items and C is the capacity given by the problem. This table is then filled in by a recurrence relation and a trace back is done in order to figure out what items were taken to produce the optimal solution. This algorithm always returns the optimal solution. The theoretical time complexity is O(n * C) for filling up the table and O(n) for the trace back. This approach was able to produce the optimal solution the fastest compared the other three methods. A drawback to this algorithm is space. For every one-byte increase needed to store the capacity, the table size increases exponentially, meaning for larger problem sets, you need much more space to store the table. A possible improvement would be to keep track of what items are taken, so that the trace back is not needed, making the program slightly faster. The main constraint of this algorithm is space. Therefore, this algorithm should be chosen in most cases, as it is the fastest one that returns the optimal solution. Avoid this algorithm if the problem size exceeds 2GB.

Branch and Bound:

The Branch and Bound approach utilizes a best first search strategy, which explores the node with the largest upper bound. The best first search strategy was implemented using a priority queue which stored the largest upper bound node at the front of the queue. The upper bound was calculated by multiplying the capacity available to the value-weight ratio of the next highest item and then adding the current value of the node. In addition to the upper bound, a lower bound was calculated as well. The lower bound was calculated by taking the value given by the greedy approach. The algorithm prunes off nodes that are infeasible (item does not fit), nodes that have an upper bound smaller than the lower bound, and nodes that have an upper bound smaller than a potential solution's upper bound. The theoretical time complexity of this approach is worst case $O(2^n)$, where n is the number of items in the problem. This algorithm did return the optimal solution, but it was much slower than the Dynamic Programming algorithm in all test cases. The Branch and Bound algorithm also suffers from space constraints as seen in the testcases with 200 items, where the program ran out of memory while running. Although this program is slow, you can stop it at any point, and it will return the best solution found so far provided that it did not run out of memory before that point. Some possible improvements to the algorithm would be to tighten the bounding function, change the search strategy to depth first search, and get a better approximate solution from the start. Removing the path attribute from each node will also improve the algorithm since it will free up more space. Therefore, this algorithm is best if you need an approximate solution in most cases.

In conclusion, all four methods are viable and the best one depends on what constraints are presented. If the algorithm needs to return an optimal solution in short amount of time with no space constraints, the Dynamic Programming solution is the best option. If you need a quick implementation with a small problem size, the Exhaustive Enumeration should be used. If you need a quick approximate solution, the Greedy approach will be the best option. Although Dynamic Programming may be the best option overall, there are cases in which the Branch and Bound approach will be faster if it can prune off multiple subtrees all at once. Considering all the tests and the implementation, I recommend using the Dynamic Programming algorithm if asked to find optimal solution, as it is the fastest and the implementation is straightforward.