KP01 – AN ANALYSIS OF ALGORITHMS

Philippe Wylezek-Serrano | pwylezek@calpoly.edu | CSC-349

1. Introduction

The following report is an analysis and detail of four separate algorithmic approaches to the 0-1 Knap Sack problem. They are as follows: Exhaustive Enumeration, Greedy, Dynamic Programming, and Branch and Bound. Each was individually tested on five distinct cases, (excluding the Full Enumeration) and recorded.
Section two contains results of the former, which can be found in **Table 1**.
Section three is a distinct analysis of each algorithmic solution, highlighting their theoretical and empirical time complexities and advantages and disadvantages of each implementation, and other pertinent details.
Section four is a conclusion containing a personal recommendation of the best algorithmic approach to take, taking into consideration all previous factors discussed in section three.

2. Experimental Results

Table 1: Empirical Time Complexities and Algorithm Results over Each Case | **LIMIT** = 300 secs

	Easy 20		**Easy 50**		**Hard 50**		**Easy 200**		**Hard 200**	
Exhaustive Enumeration	Value: 726	Weight: 519	Untested		Untested		Untested		Untested	
	Time: 4.186 sec									
Greedy	Value: 692	Weight: 476	Value: 1096	Weight: 240	Value: 16538	Weight: 10038	Value: 4075	Weight: 2634	Value: 136724	Weight: 111924
	Time: 0.005 sec		Time: 0.001 sec		Time: 0.001 sec		Time: 0.004 sec		Time: 0.001 sec	
Dynamic Programming	Value: 726	Weight: 519	Value: 1123	Weight: 250	Value: 16610	Weight: 10110	Value: 4092	Weight: 2658	Value: 137448	Weight: 112648
	Time: 0.004 sec		Time: 0.003 sec		Time: 0.017 sec		Time: 0.006 sec		Time: 0.122 sec	
Branch and Bound	Value: 726	Weight: 519	Value: 1123	Weight: 250	Value: 16610	Weight: 10110	Value: 4092	Weight: 2658	Value: 137008	Weight: 112208
	Time: 0.018 sec		Time: 0.004 sec		Time: 0.003 sec		Time: 0.061 sec		Time: **LIMIT**	

3. Analysis

Each subsection contains an analysis of each individual algorithm, including a brief description, time complexities, advantages and disadvantages, and possible improvements of each.
To preface, for each implementation, as defined by the 0-1 Knapsack problem, the algorithm will attempt to pick the item subset amongst the n given items for knapsack capacity C, whose value is maximal and total weight satisfies:

$$\sum_{i=1}^{n} w_i x_i \leq C \qquad (1)$$

$$x_i \in \{0,1\}, (i = 1, \dots , n),\ x_i = \begin{cases} 1 & \text{if selected} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

3.1. Full Enumeration

The Full Enumeration algorithm is performed by considering every possible combination of the given item set and choosing the item subset whose total value is maximal amongst sets that fit constraint defined by (1) and (2). Each combination is represented by taking each integer 0 through $2^n - 1$ and converting it to a binary string of length n. Any character x_i in said string is described by constraint (2).

For any set n of objects, there exists 2^n combination of said objects, and each objects binary string representation must be parsed to calculate its attributes, which takes n operations. Thus, the time complexity of the Full Enumeration algorithm is $\Theta(n * 2^n)$, since n operations are performed 2^n times. In practice, computing said number of operations takes an exceedingly long time for any larger n. Evident by the 4.186 second execution time (see **Table 1**) for the smallest test set, and thus only the smallest test set was attempted.

3.2. Greedy

The Greedy algorithm is performed by first sorting the list of given items in descending order of $\frac{p_i}{w_i}$, where p_i is the value and w_i is the weight of any item i. To form a solution, the algorithm continuously picks the items from the start of the list until the next item would cause the subset to no longer obey the constraints (1) and (2).

Picking is done in linear time, however, the items must be sorted. An efficient sorting algorithm is used, giving a complexity of $O(n * \log(n))$. This is a noticeable difference when inspecting the time to solution for the larger sets, notably Greedy achieves 0.001 secs, whilst others are 0.122 secs and LIMIT for the same set.

Of the four algorithms, Greedy has the best asymptotic time complexity, meaning it will yield some solution faster than the other proposed algorithms. However, this comes with the cost that the solution that Greedy generates is not optimal, which poses problems if optimal is necessary.

To improve upon this algorithm, the order in which items are selected could be altered. For example, items could be selected by greatest value first or by least weight first. However, none of these “improvements” will yield an optimal solution. A contextual consideration must be made before choosing a specific methodology.

3.3. Dynamic Programming

The Dynamic Programming algorithm is performed in two steps. The first is creating a table with dimensions $(C + 1) \times (n + 1)$, and instantiating the first column and row with zero’s. Then utilizing the recurrence relation, $V(i, j) = \max\{\ V(i - 1, j), V(i - 1, j - w_i) + \ p_i\}$, the algorithm iteratively computes cell values until each cell has been filled. The second is to call a traceback function which begins at the final cell in the table, and continuously decrements from (i = (n … 1), j = C), through a series of cell comparisons. If it determines an item to be chosen, it shifts back the comparison point by (j – w_i), then continues. Any cell (i, j) in the table is a representation of the optimal solution for a sub problem with the item subset 1, … , i, and capacity j. By utilizing previous determined optimal solutions, the table is generated as shown above.

The algorithm, for step one must compute a value for n * C cells, performing in turn n * C operations. Thus, the asymptotic time complexity is $\Theta(n * C)$. For step two, the traceback at worst must iterate through n items, and has a time complexity of $O(n)$. Thus, the overall time complexity is $\Theta(n * C)$. It is worth noting that this algorithm can produce an optimal solution for difficult subsets while still maintaining a reasonable time 0.122 secs for hard 200 set, whilst other algorithms either don’t produce an optimal solution, or cannot do so in a timely manner.

The Dynamic Programming approach is advantageous in the fact that it is guaranteed to generate an optimal solution and does so in a polynomial number of operations. However, problems exceedingly large capacities, the size of the table can become enormous, leading to issues of space complexities and memory use.

To improve the approach, implementing the Sets method could prove fruitful. The Sets approach tracks the current optimal item set, and other related sets to related to the solution, then discards unneeded sets. This method would conserve more space than the current implementation and could improve performance.

3.4. Branch and Bound

The Branch and Bound algorithm is performed by creating and pruning a state space tree (SST). In this implementation, two Node attributes are utilized in the pruning of the SST, an upper bound and a cost. The former is calculated by taking the current value of the sequence at some Node and greedily adding items values to the bound until the next item would violate constraint (1) and (2). To supplement the greedy nature, items are sorted in descending order of $\frac{p_i}{w_i}$. The latter value is calculated the same as the former, except a fraction of the constraint breaking item is taken and added to this attribute.

Both scores plus the current subset are stored in a Node object. The upper bound is a representation of the minimum bound on the best solution that may come from that Node, and the cost represents a theoretical best partial solution that may exist by expanding said Node.

The SST is explored utilizing Best-first search, supplemented by a Priority Queue, which sorts the Nodes by their cost attribute. Nodes are pruned when their cost score does not exceed the current best upper bound. This pruning technique is known as Least-Cost Upper Bound. (Bari, 2018)

This algorithm, worst case, must consider every node/state in the tree. It must also, worst case, greedily calculate every item into each bound. Thus, $O(n * 2^n)$ is the worst-case complexity. This is very noticeable on certain data sets, where an optimal cannot be reached the in under the maximum time limit (see **Table 1**).

The advantage of this algorithm implementation is that it can compete with Dynamic Programming in some instances of the problem. See times for easy 20 through hard 50 in **Table 1**. Unfortunately, the disadvantage is apparent with certain problem instances of 01KP, which can cause certain implementations to prune poorly and take extensively long times to find solutions (see **Table 1**).

The most limiting factor of any Branch and Bound algorithm is its bounding function, and any improvement would have to come through such a channel. Martello and Toth discuss in their paper an algorithm specifically designed for strongly correlated (hard) problems, but also functions more generically (Martello & Toth, 1997). Their algorithm is able to solve optimally strongly correlated problems much more efficiently and would eliminate the weakness that is described when utilizing the Least-Cost Upper Bound technique.

4. Conclusion

To conclude, I would recommend a Dynamic Programming approach. Although it does require large amounts of space to contain, which can cause issues, it can consistently and most importantly, quickly come to a solution. Full Enumeration while always yielding the correct solution, simply takes far too long to finish all its computations. Greedy, while capable of computing a solution much more quickly than Dynamic Programming, does not ensure an optimal solution. Branch and Bound, while possibly computing an optimal solution more quickly than Dynamic Programming and with lesser space, is not guaranteed to do so, and is far too dependent on its Bounding function. Thus, Dynamic Programming, all things considered, is my recommended algorithmic solution to the 0-1 Knapsack Problem.

References

Martello, S., & Toth, P. (1997). Upper Bounds and algorithms for hard 0-1 Knapsack Problems. *Operations Research*, *45*(5), 768–778. https://doi.org/10.1287/opre.45.5.768

YouTube. (2018). *7.2 0/1 Knapsack using Branch and Bound*. *Youtube*. Retrieved June 2, 2022, from https://www.youtube.com/watch?v=yV1d-b_NeK8.