Markelle Kelly – mkelly23

Knapsack Final Report

A client has requested an algorithm recommendation for the 0-1 knapsack problem. I have explored four different approaches to the problem – brute force, greedy, dynamic programming, and branch and bound – and outlined my findings below.

Enumeration

One approach to the knapsack problem is brute force: enumerating all possible combinations of objects and, from those that fit the constraints, choosing the set with the maximum value. This algorithm is easy to implement and understand, and it guarantees an optimal solution. For small problems it completes relatively quickly – e.g., less than one second for a problem with 20 items. However, this time increases exponentially as the number of items increases, making this approach unreasonable for even medium-sized problems. Additionally, memory can be an issue if all the binary strings are generated and stored at the beginning (as opposed to being generated one at a time as the algorithm runs). In fact, if implemented this way, an OutOfMemory error is thrown for even a problem of size 50.

Greedy

Another approach is to use a greedy algorithm. The effectiveness of greedy algorithms for knapsack depends on the criteria for the order in which items are added. I used the decreasing value/weight ratio for my implementation, and this has worked well. The greedy approach does not guarantee an optimal solution for the knapsack problem, but sorting by the value/weight ratio tends to give a solution that is reasonably close to the optimal solution (usually a value around 99% of the optimal solution value). Greedy is incredibly fast for problems of even very large size and is the fastest of all the algorithms tested at every size and difficulty level. It is also extremely simple to implement. Therefore, greedy is ideal if a good solution (rather than the very best solution) is acceptable.

Dynamic Programming

One of the most solid algorithms for the knapsack problem is a dynamic programming approach. Dynamic programming is very fast, completing even hard problems of size 200 in less than a second, and it guarantees an optimal solution. There are some disadvantages to dynamic programming: it is significantly more complex to implement, and it requires much more memory. This algorithm necessitates a two-dimensional table to hold partial solutions, using more storage than any of the other algorithms tested.

Branch and Bound

Finally, branch and bound can be used to solve the knapsack problem as an improvement upon the enumeration approach. Being able to prune off branches that do not fit the problem

constraints does save some time, but the speed of the algorithm is largely dependent on the bounding function used. I chose to use greedy (again with decreasing value/weight ratio) to calculate the upper bound, using a fraction of the last item added, if necessary, to fill the knapsack to capacity exactly. This is a tighter bound than many other strategies (e.g. using the best possible value/weight ratio to fill the entire knapsack) but is still relatively simple to calculate. Like enumeration, branch and bound guarantees an optimal solution, but is much more complex to implement, requiring some kind of representation of a tree and a bounding function. Branch and bound can compete with the speed of the dynamic programming approach for medium-difficulty problems, but slows down dramatically for the hard problem of size 200.

Results

value(weight), time (in ms)

	easy20	**easy50**	**hard50**	**easy200**	**hard200**
Enumeration	726(519) 382	-	-	-	-
Greedy	692(476) 0	1115(247) 2	16538(10038) 2	4090(2655) 4	136724(111924) 5
Dynamic	726(519) 1	1123(250) 2	16610(10110) 19	4092(2656) 28	137448(112648) 266
B&B	726(519) 2	1123(250) 6	16610(10110) 7	4092(2656) 40	137448(112648) 1000*

*Terminated without completing

Analysis of Results

On average, the value of the greedy solution is 98.73% of the optimal solution value. Interestingly, greedy did the worst (in this respect) on easy20, with only 95.3% of the optimal solution; the rest are all over 99%, the best being easy200 with 99.95%. For all of these, greedy is essentially completed instantaneously (5ms or less).

Of the algorithms that produce an optimal solution, dynamic is, in general, the fastest, although branch and bound was a bit faster on hard50. For problems of size 50, dynamic completion time seems much more affected by the difficulty of the problem (difference of 17ms) than branch and bound (difference of 1ms). However, for hard problems, branch and bound is much more sensitive to the size, and is much more sensitive to difficulty at larger sizes (i.e. 200 items).

All of the times recorded other than hard200 for branch and bound are under 1 second. Although branch and bound for hard200 takes a long time to terminate (overnight), it has found the optimal solution before 1 second as well.

Areas for Improvement

For brute force, if time was not a primary concern and the client wanted to use this approach for slightly larger problems, I would adjust my algorithm to fix some storage concerns.

Currently, all binary strings are generated and then iterated through; to avoid creating a large array containing all these strings, I could evaluate the weight and value for one binary string at a time as it is generated.

For greedy, the only thing that can be tweaked is the sorting of the items. Currently, I use merge sort with value/weight ratio, and this is fast and gives relatively good solutions. If it was revealed that the client's data had certain characteristics (e.g., weights are usually all the same), other sorting criteria could be considered, but for the general case, the value to weight ratio seems to give better solutions than just weight or just value.

For dynamic programming, one way to improve the algorithm would be changing to a top-down memoization approach: completing the same calculation, but recursively, and only computing the necessary subproblems. This would likely improve the algorithm's completion time and would allow for the consideration of alternative storage solutions.

For branch and bound, an exploration could be done of tighter bounding functions or special cases for pruning. While having tighter bounds speeds up branch and bound (as it increases the potential for branches to be pruned early), a bounding function that is tighter than the one I currently have implemented would be relatively complex, so the algorithm as a whole may not speed up and could even slow down.

Client Recommendations

	Easy to implement	Optimal	Fast on easy problems	Fast on medium problems	Fast on hard problems	Minimal storage use
Enumeration	X	X	X			
Greedy	**X**		**X**	**X**	**X**	**X**
Dynamic		X	**X**	X	X	
Branch & Bound		X	**X**	X		X

When to use…

- Enumeration: If the client has exclusively very small problems that only infrequently need to be solved, needs the optimal solution, and wants something straightforward and easy to implement
- Greedy: If the client has speed or low storage usage as their top priorities, and does not need the optimal solution, but just a good one; especially if they have very large problems that need to be solved frequently
- Dynamic: If the client wants an algorithm that is fast and optimal, and they do not have tight storage constraints; good for any size/frequency of problem
- Branch and Bound: If the client needs an optimal solution for small to medium size problems, has storage concerns, and wants something relatively fast

Overall, it is unlikely that enumeration or branch and bound are the best algorithm for the client; in general, if the client prioritizes speed or storage conservation over maximizing the value, they should use greedy, if their priority is optimal value over speed or storage, they should use dynamic programming.

Summary

In conclusion, the brute force algorithm for knapsack is very slow, but easy to implement. The greedy algorithm is the fastest, but is the only algorithm that does not guarantee an optimal solution. Dynamic programming is the fastest of those that do guarantee an optimal solution, but uses the most storage, and branch and bound trades some of this speed for using less storage.