Dynammic Programming

Noura and Raina

Integer programs are significantly more difficult to solve than linear programs. Linear programming is solvable in polynomial time, whereas integer programming is NP-hard and can require exponential time in the worst case. Unfortunately, many of the most interesting optimization problems fall into the second category. For example, using a brute force method, to find the optimal solution to a Knapsack Problem, it would take approximately $O(n2^n)$ time to calculate the weight and value of each subset of items. There exist approximation algorithms, such as the greedy and 2-approximation algorithms, which can significantly reduce time, but they sacrifice certainty. We examine Dynamic Programming, an algorithmic technique that is as correct as brute force but but takes much less time.

A Dynamic Programming approach only makes sense on problems with optimal substructure and overlapping subproblems. Problems with optimal substructure can be broken into smaller sub-problems that can then be used the overall solution, while problems with overlapping sub-problems are those that, with a brute force approach, would require some subproblems to be solved more than once. Problems that have optimal substructure without overlapping sub-problems are solved by the divide-and-conquer strategy, not dynamic programming. The overlapping subproblems are the source of redundancy in solving these problems with a brute force approach and that dynamic programming eliminates.

Dynamic programming is a good example of time-memory tradeoff. In order to eliminate redundant sub-problem calculations, we store their solutions. To store the solutions of the sub-problems, we have two methods: tabulation and memoization. Tabulation involves solving the sub-problems, storing the result, and moving to the next problem, while memorization is the method of computing solutions to sub-problems as they are required (on the fly).

We examine and verify the correctness of a dynamic program for solving the knapsack problem and and write a python script to automate the process. The dynamic program for the knapsack problem uses tabulation. Given a set of objects with weights and values as well as a knapsack with a weight limit to hold them, we consider the optimal solutions to each sub-problem. The subproblems, in this case, are the problems in which we have fewer items to choose from and/or a lower weight capacity. Because we start from the smallest number of items and smallest weight capacity, we can use our previous optimal solution to quickly calculate the current optimal solution. We only have to consider whether the objective value of the optimal solution increases when we take the current item. If taking the item does not increase the objective function, we simply use the objective value from the previous iteration.

Dynamic programming is effective in reducing the task-completion time of the knapsack from $O(2^n)$ to $O(nW)$ where n is the number of items and W is the maximum weight capacity of the knapsack (weights must be integers) and reduces the task-completion time of the Traveling Salesman Problem from $O(n!)$ to $O(2n)$. But Dynamic Programming has application outside of Integer programming too. Dynamic programming can be used to reduce task-completion time of computing Fibonacci terms, in matrix chain multiplication, in sequence alignment in bio-informatics, and even in solving the Tower of Hanoi Puzzle game.

Overall, this project is an introduction to the theory of dynamic programming. We studied complexity classes, and specifically, the complexities of the TSP and Knapsack problems via brute force and via Dynamic Programming, and we wrote a dynamic program for the Knapsack problem in Python.