Dynamic Programming

This document contains summary of some of the items discussed as part of the COMS 3110 Lectures. This document **does not replace the lecture materials**. This document may contain some topics that are not covered as part of the lecture; you will not be tested on those parts, they are made available to you for gaining further knowledge on topics/concepts that are related to class-lecture.

1 Topics Covered

- Dynamic programming
- Bellman-Ford algorithm
- Subset sum problem

2 Dynamic Programming

Dynamic Programming (DP) strategy is particularly suitable for efficiently solving optimization problem. This strategy is credited to Richard Bellman. The name of the strategy "dynamic programming" in Bellman's own words (from Eye of the Hurricane: An Autobiography):

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

2.1 Steps in realizing a DP algorithm

1. Identify a recursive formulation for solution of the problem. This is similar to applying divide and conquer strategy (number of subproblems to be solved may be exponential but if the number of distinct subproblems is polynomial, then DP strategy becomes effective)
2. Consider a dictionary-storage for keeping track of the result of recursive calls. The parameters of the recursion indicates the dimensions of the dictionary (one-to-one mapping).
3. Identify the dependency between the recursive calls, which, in turn, reveals the dependency between subproblems. Write an iterative computation strategy such that if subproblem P depends on the subproblem P', then the iterative strategy solves P' before P.

3 Case Study: Subset Sum Problem

We are given a set of items, where each item has a value v_i (for computing purposes, you can consider that you are given a set of numbers). We are also given a budget B. The problem is to identify a subset of the items such that sum of the value of the items in the subset is $\leq B$ and the sum is also maximal.

As we have done in the previous section, we will focus on finding the maximal sum of values that can be realized. To develop the DP solution strategy for this problem, we need to first develop a recursive characterization for the solution. Let $SS(i, b)$ represents the maximal sum of values that can be realized from the items $1 \ldots i$, when the budget is b. Therefore,

$$SS(i,b) = \begin{cases} \texttt{max}\{v_i + SS(i-1, b-v_i),\ SS(i-1,b)\} & \text{if } b - v_i >= 0 \\ SS(i-1,b) & \text{otherwise} \end{cases}$$

$$SS(i,b) = 0 \quad \text{if } i = 0 \text{ or } b = 0$$

The first condition for the first recursion corresponds to the case where the i-th item can be part of the solution when v_i does not exceed the budget b. Therefore, the solution will the maximum of the sum of the values that can be attained by either including i-th items (in which case the budget available for the rest of items is $b - v_i$) or by excluding it[1]. If the i-th item cannot be added, then the solution for $SS(i, b)$ is identical to $SS(i-1, b)$. The base case for the recursive characterization corresponds to the case when no items can be considered or no budget is available.

Based on the recursive characterization, we observe that ordering in the recursive calls requires resolving the SS-calls for smaller valuations of the parameters (i and b) before resolving the SS-calls for the larger valuations of the parameters. *If we consider the case where the values of the items and the budget are integers*, based on the above observation, we can write an iterative algorithm using a dictionary as follows:

```
// input: array V, where V[i] is the value of the item i
//        budget B
// dict[i][b]: stores the maximum sum of values within the budget b

for (i=0 to n) dict[i][0] = 0;
for (b=0 to B) dict[0][b] = 0;
for (i=1 to n)
for (b=1 to B)
if (V[i] <= b)
dict[i][b] = max{ V[i] + dict[i-1][b-V[i]], dict[i-1][b] }
else
dict[i][b] = dict[i-1][b]

return dict[n][B]
```

The runtime of the above algorithm is $O(nB)$. Though it may appear to be poly-time algorithm, note that the runtime depends on the valuation of one of the inputs (budget B) rather than the size of the input (which is $n + 1$). This algorithm is often referred to as *pseudo-polynomial*.

An accurate realization of the runtime using the size would require considering the number of bits that represent the budget B. If the number of bits representing the budget B is m, then the runtime would be $O(n2^m)$; thus revealing that the runtime is exponential with respect to the size of its input.

4 Case Study: Single Source Shortest Paths in Weighted Graph

This algorithm is due to Bellman-Ford. Consider the following functions in the context of a given source:

[1]Recall, the general strategy for all subset problem characterization is to consider two options for each item: **to include or not to include, that is the question**.

- $dist(i, v)$: shortest path distance from s to v using exactly i edges
- $sdist(i, v)$: shortest path distance from s to v using at most i edges. Therefore,

$$sdist(i, v) = \texttt{min}\{dist(0, v), dist(1, v), \ldots, dist(i, v)\}$$

- $sdist^*(v)$: shortest path distance from s to v using any number of edges. Therefore,

$$sdist^*(v) = \texttt{min}\{dist(0, v), dist(1, v), \ldots\}$$

We can write

$$\begin{aligned} sdist(i, v) &= \texttt{min}\{sdist(i-1, v),\ dist(i, v)\} \\ &= \texttt{min}(\{sdist(i-1, v)\} \ \cup\ \texttt{min}\{wt(u, v) + dist(i-1, u) \mid (u, v) \in E\}) \\ &= \texttt{min}(\{sdist(i-1, v)\} \ \cup\ \texttt{min}\{wt(u, v) + sdist(i-1, u) \mid (u, v) \in E\}) \end{aligned}$$

The last equality holds because $dist(u, i) \geq sdist(i, u)$ for all u and i. Now, we have a recursive definition for $sdist$, with the base case being

$$sdist(0, v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{cases}$$

The function $sdist^*(v)$ may not be well-defined in the presence of negative-weight cycles (cycles, where the sum of the weights of the edges is negative). This is because, one can go around a negative-weight cycle indefinitely to reduce the shortest path distance to some vertices (which will approach negative infinity).

Bellman-Ford algorithm not only identifies the correct shortest path distances in the absence of negative-weight cycles but can also identify the vertices whose shortest path distances are not well-defined when negative-weight cycles are present in the path from the source to those vertices.

The following claim holds when there is no negative weight cycle in any path from the source to the vertex v:

$$sdist^*(v) = sdist(|V| - 1, v)$$

. This claim is the basis for the Bellman-Ford algorithm. This claim holds because any path from the source to v with more than $|V| - 1$ edges will contain a cycle (based on the fact that a longest loop-free path in a graph with V vertices is $|V| - 1$). If there is no negative weight cycle in any path from source to v, then any cycle will not minimize the distance to v. That is, the shortest path distance from source to v can be computed by considering all the paths containing at most $|V| - 1$ edges ($sdist(v, |V| - 1)$). The function $sdist$ is well-defined and can be computed using DP strategy.

The recursive structure indicates that (a) dictionary is two-dimensional and (b) the shortest path distances with at most i edges must be computed before the shortest path distances with at most $i + 1$ edges. This leads to following DP strategy for computing $sdist$.

```
dict[0][s] = 0; // s is the source
dict[0][v] = infty // for all other vertices
for i=1 to |V|-1
for v in V
dict[i][v] = dict[i-1][v];
for (u,v) in E
if dict[i][v] > dict[i-1][u] + wt(u,v)
dict[i][v] = dict[i-1][u] + wt(u,v)
return dict
```

The runtime of the algorithm is $O(|V||E|)$.

Presence of negative-weight cycles. The corollary to the claim that led to Bellman-Ford is: When there is a negative weight cycle in the path from source to a vertex v, then the shortest path distance for v is not equal to $sdist(v, |V| - 1)$. A stronger claim can be made: if there is a negative weight cycle, then there exists at least one vertex v, for which $sdist(v, |V| - 1) > sdist(v, |V|)$. This is because if there is a negative weight cycle, then there exists at least one vertex that participates in that cycle and its shortest path distance becomes smaller after the cycle is explored one time (path containing at most $|V|$ edges indicates the cycle is explored at least one time in the path). Such a vertex can be identified by running the outermost for-loop of Bellman-Ford algorithm one more time and checking whether the dictionary entries in the $|V|$-row is identical to that in $|V| - 1$ row; if not, then there is a negative weight cycle in the graph. Also, one can identify the vertex (column in the dictionary) or vertices whose dictionary entry in $|V|$ row is smaller than that in $|V| - 1$ row; the shortest distances from source to these vertex (or vertices) are not well-defined due to the presence of negative weight cycles. The same is true for all other vertices that are reachable from these vertices; that is, simple BFS or DFS will reveal all vertices whose shortest path distances are not well-defined.

Side notes. One can compute the path that realizes the shortest path distance from source vertex by using dictionary entries. One can also add a parent information to vertex v and update that information as `dict[i][v]` gets updated.

The algorithm can be optimized to stop early when the i-th row and $i-1$-th row have the same valuations in the dictionary. The algorithm can be also optimized by considering the for-loop over vertices and its neighbors as a single for-loop over all the edges.

These optimizations do not impact the runtime of the algorithm.

Think about whether the algorithm can be space optimized. Is it necessary to maintain the dictionary of size $O(|V|^2)$?

5 Exercise

1. Given a dynamic array of positive integers, `coins`, representing coin denominations and an integer `amount`, return the minimum number of coins needed to make up that amount. If it's not possible, return -1.

2. Given a dynamic array of integers, `nums`, return the length of the longest strictly increasing subsequence.

3. Given a dynamic array of integers, `nums`, find the contiguous subarray (containing at least one number) which has the largest product, and return the product. Note that `nums` may contain a negative numbers and zeros, which can flip the sign of the product or reset it to zero.

4. A robot can move only right or down in an $m \times n$ grid. Given integers m and n, return the number of unique paths from top-left to bottom-right.

5. Given a string `s` and a dictionary `wordDict`, return True if `s` can be segmented into a space-separated sequence of dictionary words.