CS 161 (Stanford, Winter 2024) Homework 6

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we look for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

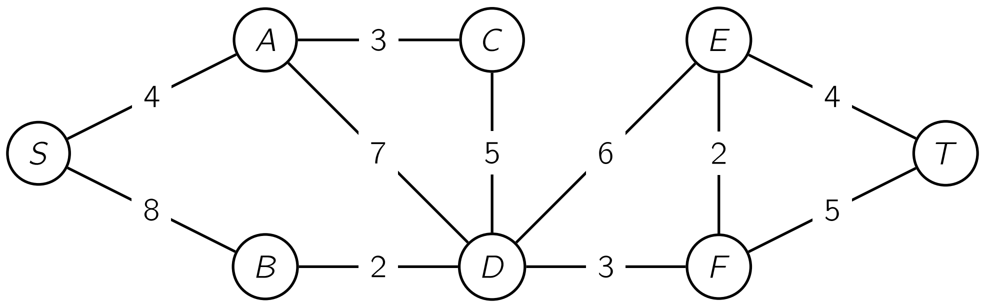
Exercises. The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

1 Exercise: Shortest Paths

1.1 Dijkstra Basics (4 pt.)

Given the undirected graph below, run Dijkstra's Algorithm from vertex S. What is the order of vertices that we will visit? What are the shortest distances from vertex S to all the other vertices?

Fill in the table below **in the order of visited vertices** and the corresponding shortest distance from vertex S to each vertex. The first column is already filled for you.

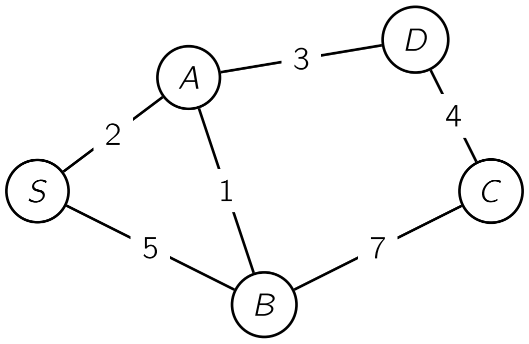


[We are expecting: Replace every "??" in the table below. No explanation is required.**]**

vertex	S	??	??	??	??	??	??	??
shortest distance	0	??	??	??	??	??	??	??

1.2 Bellman-Ford Basics (4 pt.)

Given the undirected graph below, run the Bellman-Ford Algorithm from the lecture to find the shortest distances from vertex S to other vertices. Fill in the table below which keeps track of the shortest distances. The first row is already filled for you.



[We are expecting: Replace every "??" in the table below. No explanation is required.**]**

-	S	A	B	C	D
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	??	??	??	??
$d^{(2)}$	0	??	??	??	??
$d^{(3)}$	0	??	??	??	??
$d^{(4)}$	0	??	??	??	??

2 Exercise: Longest Paths

For this problem, let $G = (V, E)$ be a weighted[1] directed acyclic graph (DAG) with n vertices and m edges. In this problem you will design a dynamic programming algorithm to find the length of a *longest* path in G.

You may also assume that for vertices v_i, v_j, you can access the edge weight via this notation: $w(v_i, v_j)$. If you want to access edge weights in your pseudocode for part (b), you may assume that fetching the weight of an edge can be done in time $O(1)$.

2.1 Identify optimal sub-structure and a recursive relationship (3 pt.)

Suppose that $v_0, v_1, \ldots, v_{n-1}$ is a topological ordering of the vertices in V. We'll define the subproblems for you. For $k \in \{0, \ldots, n-1\}$, let $P[k]$ denote the length of the *longest* path in G which starts at any vertex and ends at the vertex v_k. (Here, "longest" means highest-cost, according to the edge weights).

State a recursive formulation which defines $P[k]$ in terms of $P[j]$ for $0 \le j < k$.

In case you have questions about how to best format your recursive formulation, one **incorrect** formula that is at least formatted appropriately is $P[k] = \min(\{P[i] + P[j] : (i, j) \in E\})$. This is incorrect for a number of reasons, including the fact that i and j are not necessarily strictly less k, which violates a requirement for this part.

[1]Edge weights could possibly be negative.

[We are expecting: A formula of the form "$P[k]$ = (something which may reference $P[0], P[1], \ldots, P[k-1]$, and the structure of G)," and base case(s). You don't need to explain why your formula is correct.**]**

2.2 Develop a DP algorithm to find the value of the optimal solution (5 pt.)

Develop a dynamic programming algorithm (either bottom-up or top-down, your choice) that uses your relationship from part (a) to find the *length* of the longest path in G. Your algorithm should take as input the DAG G, with the vertices ordered in a topological ordering, and output the length of the longest path in G (Here, "longest" again means highest-cost, according to the edge weights).

Your algorithm should run in time $O(n+m)$. You may assume that the vertices are stored in an adjacency-list format so that `V[i].inNeighbors()` returns a list of the indices $j \in \{0, \ldots, n-1\}$ so that $(v_j, v_i) \in E$. Remember, you may access the weight of an edge (v_i, v_j) using $w(v_i, v_j)$, which runs in time $O(1)$.

[We are expecting: Pseudocode. You **do not** need to include an English explanation, although you may if you think it will make your answer more clear. You **do not** need to justify the running time.**]**

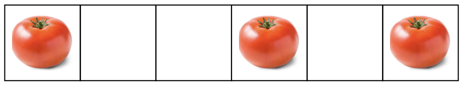
Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

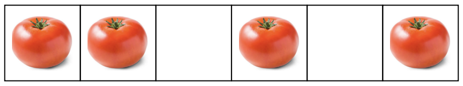
3 Rotten Tomatoes

You are planting tomato plants in a garden, and the garden has n spots arranged in a line. Different spots in the garden will result in different quality tomatoes: suppose that the location i will result in tomatoes of deliciousness $T[i]$, where $T[i]$ is a positive integer. Further, you cannot plant two plants directly next to each other, because they will compete for resources and wilt. Your goal is to create the most deliciousness possible (summed up over all of the tomato plants).

For example, if the input was $T = [21, 4, 6, 20, 2, 5]$, then you should plant tomatoes in the pattern



and you would obtain deliciousness $21 + 20 + 5 = 46$. You would **not** be allowed to plant tomatoes in the pattern



because there are two tomato planted next to each other.

In this question, you will design a dynamic programming algorithm which runs in time $O(n)$ which takes as input the array T and returns the maximum deliciousness possible given T. Do this by answering the two parts below.

3.1 Identify optimal sub-structure and a recursive relationship (6 pt.)

What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between the sub-problems?

[We are expecting:

- A clear description of your sub-problems.
- A recursive formulation that they satisfy, along with a base case.
- An informal justification that the recursive relationship is correct.

]

3.2 Develop a DP algorithm to find the value of the optimal solution (6 pt.)

Write pseudocode for your algorithm. Your algorithm should take as input the array T, and return a single number which is the maximum amount of deliciousness possible, in time $O(n)$. Your algorithm does not need to output the optimal way to plant the tomatoes.

[We are expecting: Pseudocode **AND** a clear English description. You do not need to justify that your algorithm is correct, but correctness should follow from your reasoning in part (a).**]**

4 Fast All-Pairs Shortest Paths

Suppose our graph is a directed graph. In this problem you will design an algorithm for the (possibly negative) weighted All-Pairs Shortest Path problem that is faster than Floyd-Warshall on sparse graphs. The main idea is to increase the weight of negative edges until all the weights are positive, and then run Dijkstra n times (i.e., from every vertex).

4.1 (5 pt.)

Let $-M$ be the smallest edge weight, so that if any weights are negative, M is positive. Consider the following naïve algorithm:

```
for all e in E:
    w'(e) = w(e) + M // w' is always non-negative
for all v in V:
    // run Dijkstra from v on modified weights.
    // ShortestPathTree is a mapping data structure that can be indexed
    // as (v, u) to recover the tree path v -> u
    ShortestPathTree = Dijkstra(V,E,w')
    for all u in V:
        // recompute distances with original weights
        dist(v,u) = ShortestPathTree(v,u).length - M * ShortestPathTree(v,u).numEdges
```

Explain why the naïve algorithm doesn't work.

[We are expecting: A counter example and a short explanation**]**

4.2 (6 pt.)

Here is a clever way to change the weights of the graph: suppose that we design a function h associating a numerical value to each vertex $v \in V$, referred to as *potential function*. We can change the weight of edge (u, v) with the formula:

$$w'(u, v) := w(u, v) + h(u) - h(v).$$

Argue that for every pair s, t of vertices, the shortest path from s to t with the new weights w' is the same as the shortest path with the old weights.

[We are expecting: A succinct but rigorous proof.**]**

Hint: Write down an expression for the new vs old weight of path $s, v_1, \ldots, v_k, t$.

4.3 (6 pt.)

Here is one way of finding a good potential: add a dummy vertex q to the graph, connect it to all other vertices with cost = 0 edges, and compute the shortest path from q to all other vertices.

Argue that if we use the shortest distance from q to each vertex as a potential function, then w' defined in the previous part is always non-negative.

[We are expecting: A succinct but rigorous proof.**]**

4.4 (12 pt.)

Design an algorithm for weighted all-pairs shortest path in a graph with negative edges but no negative cycles. Suppose $m = \Omega(n)$, your algorithm should run in time $O(nm\log(n))$ (or a little faster using Fibonacci heaps).

[We are expecting: A clear English description, pseudocode, and a short justification of runtime. The pseudocode can use the algorithm(s) from the lecture as black-box function(s).**]**

5 Taking Stock

Suppose you have access to trustworthy, confidential information regarding the future prices of k different stocks over an upcoming period of n days. That is, for each day $i \in [1, n]$ and each stock $j \in [1, k]$, you're given $p_{i,j}$, the trading price of stock j on day i. You start with a budget of P dollars, and on each day, you may buy or sell any number of shares from any stock each day, as long as you can afford them. (Assume that all prices are integer-valued and that you can only purchase whole stocks.) You have an earning goal of Q dollars, so your profit goal is $Q - P$. Here, we will design an algorithm to determine whether you can meet your goal, and if not, the maximum profit achievable.

5.1 Two days (8 pt.)

Suppose we are only looking at prices over two days (i.e. $n = 2$). Design an $O(kP)$ dynamic programming algorithm that computes the maximum amount of profit you can make buying stocks on the first day and selling stocks on the second day. Prove the runtime and correctness of your algorithm.

Hint: Let $M[l]$ be the maximum amount of profit you can make by buying l dollars of stock on the first day. Write a recursive relationship for $M[l]$.

[We are expecting: Pseudocode or a clear English description of the algorithm, a runtime analysis, and a rigorous proof of correctness**]**

5.2 *N* days (8 pt.)

Now, suppose you are given prices over n days. Using your solution to part (a) as a guide, design an $O(nkQ)$ time algorithm that determines whether you can reach your goal, and if not, reports the maximum profit achievable. Prove your algorithm's runtime and correctness.

Hint: It's helpful to reframe each day as 1) selling all the shares you own and 2) then buying a set of shares that you can afford.

[We are expecting: Pseudocode or a clear English description of the algorithm, a runtime analysis, and a rigorous proof of correctness**]**