CS/ECE 374 ✧ Fall 2018

Homework 8 (Updated on Nov 2, 2:03PM)

Due Wednesday, November 7, 2018 at 10am

Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

The following unnumbered problems are not for submission or grading. No solutions for them will be provided but you can discuss them on Piazza.

- In the lab you saw how to compute s-t shortest walks efficiently when the graph has a single negative length edge. The running time is asymptotically the same as using Dijkstra's algorithm. Generalize this to the setting where the graph has *two* negative length edges. Can you generalize the approach to any fixed number of edges? If there are k negative length edges what would be the running time of your algorithm? Is this better than using Bellman-Ford?
- Given a directed graph $G = (V, E)$ with non-negative edge lengths, and two nodes s, t, the bottlenect length of a path P from s to t is the maximum edge length on P. The bottleneck distance from s to t is defined to be the smallest bottleneck path length among all paths from s to t. Describe an algorithm to compute the bottleneck shortest path distances from s to every node in G by adapting Dijkstra's algorithm. Can you also do it via a reduction to the standard shortest path problem?
- See problems in HW 8 from Spring 2018. https://courses.engr.illinois.edu/cs374/sp2018/A/homework/hw8.pdf

1. Graphs are a powerful tool to model many phenomena. The edges of a graph model pairwise relationships. It is natural to consider higher order relationships. Indeed *hypergraphs* provide one such modeling tool. A hypergraph $G = (V, \mathcal{E})$ consists of a finite set of nodes/vertices V and a finite set of hyper-edges $\mathcal{E}$. A hyperedge $e \in \mathcal{E}$ is simply a subset of nodes and the cardinality of the subset can be larger than two. An undirected graph is a hypergraph where each hyper-edge is of size exactly two. Here is an example. $V = \{1, 2, 3, 4, 5\}$ and $\mathcal{E} = \{\{1, 2\}, \{2, 3, 4\}, \{1, 3, 4, 5\}, \{2, 5\}\}$. The representation size of a hypergraph is $|V| + \sum_{e \in \mathcal{E}} |e|$. An alternating sequence of nodes and edges $x_1, e_1, x_2, e_2, \ldots, e_{k-1}, x_k$ where $x_i \in V$ for $1 \leq i \leq k$ and $e_j \in \mathcal{E}$ for $1 \leq j \leq k-1$ is called a path from u to v if (i) $x_1 = u$ and $x_k = v$ and (ii) for $1 \leq j < k$, $x_j \in e_j$ and $x_{j+1} \in e_j$.

    - Given a hypergraph $G = (V, \mathcal{E})$ and two nodes $u, v \in V$ we say that u is connected to v if there is path from u to v. We say that a hypergraph is connected if each pair of nodes u, v in G are connected. Describe an algorithm that given a hypergraph G

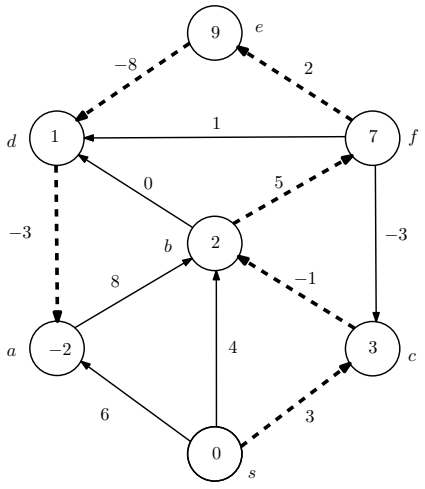
checks whether G is connected in linear time. In essense describe a reduction of this problem to the standard graph connectivity problem. You need to prove the correctness of your algorithm.

- Suppose we want to quickly spread a message from one person to another person during an emergency on a social network called AppsWhat which is organized as a collection of groups. Messages sent by a group member are broadcast to the entire group. AppsWhat knows the members and the list of groups on its service. The goal is to find the fewest messages that need to be sent such that a person u can reach a person v. Model this problem using hypergraphs and describe a linear-time algorithm for it. No proof necessary for this part.

2. Let $G = (V, E)$ be a directed graph with edge lengths that can be negative. Let $\ell(e)$ denote the length of edge $e \in E$ and assume it is an integer. Assume you have a shortest path tree T rooted at a source node s that contains all the nodes in V. You also have the distance values $d(s, u)$ for each $u \in V$ in an array (thus, you can access the distance from s to u in $O(1)$ time). Note that the existence of T implies that G does not have a negative length cycle.

   - Let $e = (p, q)$ be an edge of G that is *not* in T. Show how to compute in $O(1)$ time the smallest integer amount by which we can decrease $\ell(e)$ before T is not a valid shortest path tree in G. Briefly justify the correctness of your solution.
   - Let $e = (p, q)$ be an edge in the tree T. Show how to compute in $O(m + n)$ time the smallest integer amount by which we can increase $\ell(e)$ such that T is no longer a valid shortest path tree. Your algorithm should output ∞ if no amount of increase will change the shortest path tree. Briefly justify the correctness of your solution.

   The example below may help you. The dotted edges form the shortest path tree T and the distances to the nodes from s are shown inside the circles. For the first part consider an edge such as (b, d) and for the second part consider an edge such as (f, e).



3. Since you are taking an algorithms class you decided to create a fun candy hunting game for Halloween. You set up a maze with one way streets that can be thought of as a directed graph $G = (V, E)$. Each node v in the maze has $w(v)$ amount of candy located at v.

- Each of your friends, starting at a given node s, has to figure out the maximum amount of candy they can collect. Note that candy at node v can be collected only once even if the node v is visited again on the way to some other place.
- Your friends complain that they can collect more candy if they get to choose the starting node. You agree to their their request and ask them to maximize the amount of candy they can collect starting at any node they choose.

Before you ask your friends to solve the game you need to know how to do it yourself! Describe efficient algorithms for both variants. Ideally your algorithm should run in linear time. *Hint:* Consider what happens if G is strongly connected and if it is a DAG.

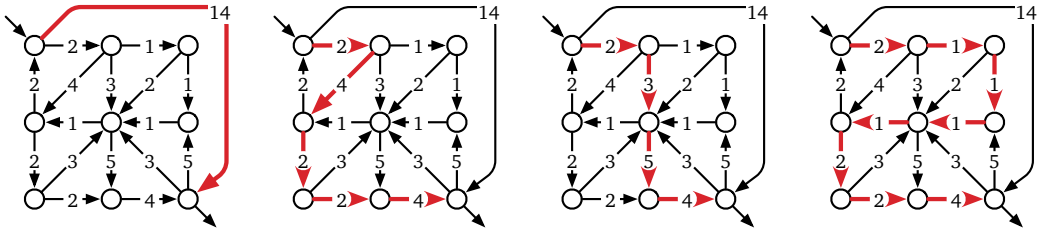
No proof necessary if you use reductions to standard algorithms via graph transformations and simple steps. Otherwise you need to prove the correctness.

4. **Not to submit but strongly encouraged to solve:** Let $G = (V, E)$ a directed graph with non-negative edge lengths. Let $R \subset E$ and $B \subset E$ be red and blue edges (the rest are not colored). Given s, t and integers h_r and h_b describe an efficient algorithm to find the length of a shortest s-t path that contains at most h_r red edges and at most h_b blue edges.

5. **Not to submit but strongly encouraged to solve:** Read the notes to see the connection between DP and DAGs. Solve the McKing problem from HW 7 via a reduction to a shortest path problem on DAGs without invoking DP.

Solved Problem

4. Although we typically speak of "the" shortest path between two nodes, a single graph could contain several minimum-length paths with the same endpoints.



Four (of many) equal-length shortest paths.

Describe and analyze an algorithm to determine the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in $O(1)$ time.

[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What's left?]

Solution: We start by computing shortest-path distances $dist(v)$ from s to v, for every vertex v, using Dijkstra's algorithm. Call an edge $u \to v$ ***tight*** if $dist(u) + w(u \to v) = dist(v)$. Every edge in a shortest path from s to t must be tight. Conversely, every path from s to t that uses only tight edges has total length $dist(t)$ and is therefore a shortest path!

Let H be the subgraph of all tight edges in G. We can easily construct H in $O(V + E)$ time. Because all edge weights are positive, H is a directed acyclic graph. It remains only to count the number of paths from s to t in H.

For any vertex v, let $PathsToT(v)$ denote the number of paths in H from v to t; we need to compute $PathsToT(s)$. This function satisfies the following simple recurrence:

$$PathsToT(v) = \begin{cases} 1 & \text{if } v = t \\ \displaystyle\sum_{v \to w} PathsToT(w) & \text{otherwise} \end{cases}$$

In particular, if v is a sink but $v \neq t$ (and thus there are no paths from v to t), this recurrence correctly gives us $PathsToT(v) = \sum \varnothing = 0$.

We can memoize this function into the graph itself, storing each value $PathsToT(v)$ at the corresponding vertex v. Since each subproblem depends only on its successors in H, we can compute $PathsToT(v)$ for all vertices v by considering the vertices in reverse topological order, or equivalently, by performing a depth-first search of H starting at s. The resulting algorithm runs in $O(V + E)$ time.

The overall running time of the algorithm is dominated by Dijkstra's algorithm in the preprocessing phase, which runs in $\boldsymbol{O(E \log V)}$ ***time***. ■

Rubric: 10 points = 5 points for reduction to counting paths in a dag + 5 points for the path-counting algorithm (standard dynamic programming rubric)