# HW 8: Extra problems

Instructors: Chan, Hassanieh, Har-Peled

**CS/ECE 374: Algorithms & Models of Computation, Spring 2019** Version: **1.0**

**1** 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.

**2** See HW 8 problems from Fall 2016 available at `https://courses.engr.illinois.edu/cs374/fa2016/homework/hw8.pdf`.

**3** 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 legnth 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?

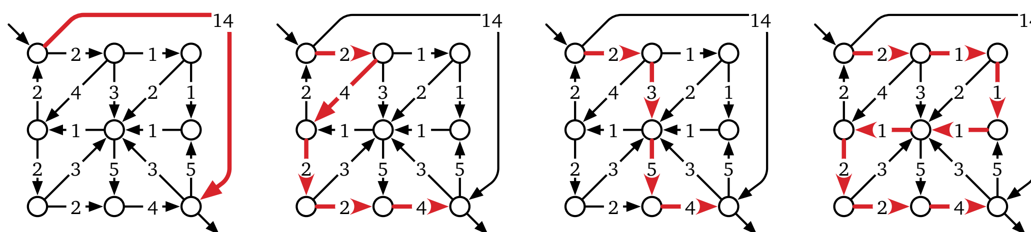
**4** Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let $s$ and $t$ be vertices of $G$, and let $H$ be a subgraph of $G$ obtained by deleting some edges. Suppose we want to reinsert exactly one edge from $G$ back into $H$, so that the shortest path from $s$ to $t$ in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert. Ideally the running time of your algorithm should be *asymptotically* the same as that of running Dijkstra's algorithm.

**5** Let $G = (V, E)$ be a directed graph. Describe a linear-time algorithm that given $G$, a node $s \in V$ and an integer $k$ decides whether there is a *walk* in $G$ starting at $s$ that visits at least $k$ distinct nodes. The following questions may help you.

- What is the answer if $G$ is strongly connected?
- How would you solve the problem if $G$ is a DAG?

**6** 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.

**7** (Hard.) 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 is 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)

**8** After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Champaign-Urbana is currently suffering from a plague of zombies, so even though the bus stops have fences that *supposedly* keep the zombies out, you'd still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between buses at least once.

Describe and analyze an algorithm to determine a sequence of bus rides from Siebel to your home, that minimizes the total time you spend waiting at bus stops. You can assume that there are $b$ different bus lines, and each bus stops $n$ times per day. Assume that the buses run exactly on

schedule, that you have an accurate watch, and that walking between bus stops is too dangerous to even contemplate.

**9** Kris is a professional rock climber (friends with Alex and the rest of the climbing crew from HW6) who is competing in the U.S. climbing nationals. The competition requires Kris to use as many holds on the climbing wall as possible, using only transitions that have been explicitly allowed by the route-setter.

The climbing wall has $n$ holds. Kris is given a list of $m$ pairs $(x, y)$ of holds, each indicating that moving directly from hold $x$ to hold $y$ is allowed; however, moving directly from $y$ to $x$ is not allowed unless the list also includes the pair $(y, x)$. Kris needs to figure out a sequence of allowed transitions that uses as many holds as possible, since each new hold increases his score by one point. The rules allow Kris to choose the first and last hold in his climbing route. The rules also allow him to use each hold as many times as he likes; however, only the first use of each hold increases Kris's score.

1. Define the natural graph representing the input. Describe and analyze an algorithm to solve Kris's climbing problem if you are guaranteed that the input graph is a dag.
2. Describe and analyze an algorithm to solve Kris's climbing problem with no restrictions on the input graph.

Both of your algorithms should output the maximum possible score that Kris can earn.

**10** Many years later, in a land far, far away, after winning all the U.S. national competitions for 10 years in a row, Kris retired from competitive climbing and became a route setter for competitions. However, as the years passed, the rules changed. Climbers are now required to climb along the *shortest* sequence of legal moves from one specific node to another, where the distance between two holds is specified by the route setter. In addition to the usual set of $n$ holds and $m$ valid moves between them (as in the previous problem), climbers are now told their start hold $s$, their finish hold $t$, and the distance from $x$ to $y$ for every allowed move $(x, y)$.

Rather than make up this year's new route completely from scratch, Kris decides to make one small change to last year's input. The previous route setter suggested a list of $k$ new allowed moves and their distances. Kris needs to to choose the single edge from this list of suggestions that decreases the distance from $s$ to $t$ as much as possible.

Describe and analyze an algorithm to solve Kris's problem. Your input consists of the following information:

- A directed graph $G = (V, E)$.
- Two vertices $s, t \in V$.
- A set of $k$ new edges $E'$, such that $E \cap E' = \varnothing$
- A length $\ell(e) \geq 0$ for every edge $e \in E \cup E'$.

Your algorithm should return the edge $e \in E'$ whose addition to the graph yields the smallest shortest0path distance from $s$ to $t$.

For full credit, your algorithm should run in $O(m \log n + k)$ time, but as always, a slower correct algorithm is worth more than a faster incorrect algorithm.

## 11 (100 PTS.) Flood it.

(This question was inspired by the game *Open Flood* [available as an app on android].)

You are given a directed graph $\mathsf{G}$ with $n$ vertices and $m$ edges (here $m \geq n$). Every edge $\mathsf{e} \in \mathsf{E}(\mathsf{G})$ has a color $c(\mathsf{e})$ associated with it[1]. The colors are taken from a set $C = \{1, \ldots, \xi\}$ (assume $\xi \leq n$), and every color $c \in C$, has price $p(c) > 0$.

Given a start vertex $s_0 = s$, and a sequence of $\Pi = \langle c_1, \ldots, c_\ell \rangle$ of colors, a ***compliant walk***, at the $i$th time, either stays where it is (i.e., $s_i = s_{i-1}$), or alternatively travels a sequence of edges of color $c_i$ that starts at $s_i$. Formally, if there is a path $\sigma \equiv (u_1, u_2), (u_2, u_3), \ldots, (u_{\tau-1}, u_\tau) \in \mathsf{E}(\mathsf{G})$, such that $c\big((u_j, u_{j+1})\big) = c_i$, for all $j$, and $u_1 = s_i$, then one can set $s_{i+1} = u_\tau$. The ***price*** of $\Pi$ is $p(\Pi) = \sum_{i=1}^{\ell} p(c_i)$.

Describe an algorithm, as fast as possible, that computes the cheapest sequence of colors for which there is a compliant walk in $\mathsf{G}$ from a vertex $s$ to a vertex $t$.

For full credit, your algorithm should run in $O(m \log m)$ time (be suspicious if you get faster running time). Correct solutions providing polynomial running time would get 50% of the points.

## 12 (100 PTS.) Flood it II.

We use the same framework as the previous question. Describe an algorithm, as fast as possible, that given a vertex $s$, and vertices $t_1, \ldots, t_k$, computes the cheapest sequence $\Pi$ of colors for which there are $k$ walks $W_1, \ldots, W_k$, such that $W_i$ is compliant with $\Pi$, and it walks from $s$ to $t_i$, for $i = 1, \ldots, k$.

Do not use dynamic programming here [it doesn't work]. Instead, think about the state of the system after $i$ colors of the sequence were used – where would the $k$ walks in the graph be at this point in time? Build the appropriate state graph, and solve the appropriate problem in this graph. The running time of your algorithm should be polynomial if $k$ is a constant. As usual, you might want to start thinking about the case $k = 2$ first.

You should be able to solve this question fully even if your solution to the first question is sub-optimal.

## 13 (100 PTS.) Halloween!

It is Halloween, and Zaphod Beeblebrox is ready. He has a map (i.e., directed graph) of Shampoo-Banana with the houses marked, with each house $v$ marked with how much candy $c(v) \geq 0$ one gets if visiting this house. The map also connects houses $u, v$ by a directed edge $(u, v)$, if Zaphod is willing to go directly from $u$ to $v$. For reasons that are not well understood (maybe, Zaphod's brain-care specialist, Gag Halfrunt knows why), the fact that Zaphod is willing to go from $u$ to $v$, does not imply that he is willing to go from $v$ to $u$ (i.e., the graph is truly a directed graph).

Given the directed graph $\mathsf{G}$, describe an algorithm as fast possible (and prove its correctness), that computes the walk that starts from a vertex $s$, and collects as much candy as possible. The algorithm should only output the amount of candy the optimal walk collects.

Note, that the walk is allowed to visit the same vertex several times, but you get candy only the first time you visit it.

[1]This is a simple directed graph – no self loops or parallel edges. Every edge has only a single color associated with it. (Unless explicitly stated otherwise, you can always assume a given directed graph is simple.)

Also, show how to modify the algorithm so that it outputs the optimal walk. What is the length of this walk in the worst case?[2]

[Hint: What if the graph is strongly connected?]

[2]There was extensive discussion on my neighborhood mailing list after Halloween with various statistics [how many kids visited, what candies were consumed, etc] – people seems to be taking this stuff seriously.