CS 330, Fall 2025, Homework 6

Due: Wednesday, 10/22, at 11:59 pm on Gradescope

Collaboration policy If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a **violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.**

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise (and concise) description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g., correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

David Bunger, Collaborators: None

Note that this homework has one written and one programming assignment. Make sure to complete both!

1. Pretty Good Navigation

You are the CEO of a popular navigation app. Currently, if a driver wants to get from some city A to another city B, the navigation app always recommends the same shortest path between these cities. (Don't forget that the definition of shortest paths is in terms of distance and not the number of edges.) Unfortunately, this leads to a lot of traffic on these shortest routes. To reduce traffic, you decide to update your app to instead recommend one of several paths such that their lengths is within a certain range of the true distance. We refer to the shortest paths as "short" and the other acceptable paths as "kind of short".

In this problem, you will design an algorithm that determines how many *short* and *kind of short* paths exist between cities. The road network is modeled as a weighted, directed graph G, where edge weights represent distances between nodes.

- (Do not hand in) The presence of edges with weight 0 can make the problem ill-defined; that is, there may not be a valid answer to the problem. Explain why.

To simplify the problem, assume that **all edge weights are positive integers and that every edge weight is at least 2**. Specifically, you want to write an algorithm that does the following:

- Takes as input the directed weighted graph G and a start city s (i.e., source node). *(You may assume that G is given to you as a weighted adjacency list in the nested hash table format.)*
  - A path from s to v is *short* if its length is the minimum distance $dist[v]$ from s to v.
  - A path from s to v is *kind of short* if its length is $dist[v] + 1$.
- Returns for each city the *number* of *short* paths and *kind of short* paths from s. *(You may return this information in the data structure of your choice provided that it's easy to identify the two numbers for each city.)*

You may use the following without proof: Consider a path $p = [s, \dots, u, v]$ from s to v whose last edge (u, v) has weight ≥ 2. If p is *short*, then $dist_s(u) \leq dist_s(v) - 2$. If p is *kind of short*, then $dist_s(u) \leq dist_s(v) - 1$.

Algorithm 1: PrettyGoodNavigation $(G,\ s)$

```
/* A modified version of Dijkstra's Algorithm that keeps track of short and kind of short
   paths as it progresses.  It also doesn't keep a parents tree, as this isn't needed     */
1  $\pi \leftarrow \{\}$
2  $paths \leftarrow \{\}$/* creates a list to store tuples in the format (number of short paths, number
   of kind of short paths)                                                                */
3  $dist \leftarrow \{\}$
4  $Q \leftarrow PQ$
5  $\pi[s] = 0$
6  $paths[s] = (0, 0)$
7  $Q.INSERT(< 0, s >)$
8  for $v \neq s$ in $G$ do
9      $\pi[v] = \infty$
10     $Q.INSERT(< \pi[v], v >)$
11     $paths[v] = (0, 0)$/* sets the number of paths for all nodes to (0,0)               */
12 while $Q$ is not empty do
13     $< \pi[u], u > \leftarrow EXTRACT - MIN(Q)$
14     $dist[u] \leftarrow \pi[u]$
15     for $v$ in $G[u]$ do
16         if $\pi[v] == dist[u] + G[u][v]$ then
17             $paths[v][0] + +$/* If the current path to v is the same as the current shortest
                 path to v, increments the short path counter for v                       */
18         if $\pi[v] > dist[u] + G[u][v]$ then
19             if $\pi[v] - (dist[u] + G[u][v]) == 1$ then
20                 $paths[v][1] = paths[v][0]$/* If the current path to v is shorter than the
                     current shortest path to v, check if it is shorter by 1.  If so, set the
                     kind of short path counter for v to be equal to the short path counter */
21             else
22                 $paths[v][1] = 0$
               /* If the current path to v is shorter than the current shortest path by more
                 than 1, set the kind of short path counter for v to 0                    */
23             $paths[v][0] = 1$/* Sets the short path counter for v to 1                */
24             $DECREASE - KEY(< dist[u] + G[u][v], v >, < \pi[v], v >)$
25             $\pi[v] \leftarrow dist[u] + G[u][v]$
26         if $dist[u] + G[u][v] == \pi[v] + 1$ then
27             $paths[v][1] + +$/* If the current distance to v is equal to its current shortest
                 distance plus 1, increments the kind of short path counter for v         */
28 return paths
```