CS 31: Homework 8

Spring 2025

Due: Wednesday, May 28, 2025 by 11:59 PM

General Instructions

- Write rigorous, clear, and concise answers. Latex your answers, or be sure your handwriting and the pdf of your scan are easy to read.
- I recommend using latex, which you can download free. Because latex has a steep learning curve, you should start learning it right away if you plan to use it. You may of course typeset using other methods, such as Word and Equation Editor.
- Submit your solutions (as a single pdf file) online using the Canvas system.

1. **Kosaraju's in Detail** (10 points)

   Let $G = (V, E)$ be a directed graph with $V = \{1, 2, \ldots, n\}$, and k be the number of SCCs of G. One way to represent the SCCs of G is with an array S of length n such that, for all $i, j \in \{1, 2, \ldots, n\}$, $S[i] \in \{1, 2, \ldots, k\}$ and $S[i] = S[j]$ if and only if vertices i and j are in the same strongly connected component of G.

   Write an algorithm of $\Theta(n+m)$ time and space complexity that, given the adjacency list representation of a directed graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, returns (k, S) where k is the number of SCCs of G and S is an array representing the SCCs of G.

   You don't need to prove your algorithm correct. You don't need to analyze its time complexity, but make sure its time complexity is $\Theta(n + m)$.

   Here are some helpful hints/notes/requirements:

   - We have done the SCC algorithm in class, but I want you to rewrite it, fleshing out the details. In particular, instead of simply saying "Do DFS of G" or "Do DFSvisit(u)", I want you to actually include the necessary code (still only at a high level in pseudocode). Keep only those details of DFS that are relevant to the problem. For instance, there may be no need for certain colors or times; don't record any unnecessary information.
   - If you need a stack or a queue, your code can simply say "let A be an empty stack" and later insert an item e by "push(A, e)".
   - You can assume that the method *transpose*(G) returns the adjacency list of G^T in $\Theta(V(G)+E(G))$ time. Recall that you wrote this method in last week's homework.

2. **Simpler SCC** (5 points)

   Dr. Murkha claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of *increasing* finishing times. Does this simpler algorithm always produce correct answers? Justify your answer.

3. **Component Graph** (10 points)

   Write an algorithm of $\Theta(n+m)$ time and space complexity that, given
   (i) the adjacency list A of a directed graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, and
   (ii) the array S that represents the SCCs of G,
   returns the adjacency list of the graph G^{SCC}, which I define below exactly as in class:

   $$G^{SCC} = (V^{SCC}, E^{SCC}),$$

   where $V^{SCC} = \{C \mid C \text{ is an SCC of } G\}$ and $E^{SCC} = \{(C, C') \mid C \neq C', \text{ and } \exists u \in C, u' \in C', (u, u') \in E\}$

   You don't need to prove your algorithm correct. You don't need to analyze its time complexity, but make sure its time complexity is $\Theta(n+m)$. (An important aspect of this problem is ensuring that the algorithm does not add the same edge more than once in the adjacency list of G^{SCC}, while still running within $\Theta(n+m)$ time.)

4. **Semiconnectivity** (10 points)

   A directed graph $G = (V, E)$ is *semiconnected* if, for all pairs of vertices $u, v \in V$, there is a path from u to v or there is a path from v to u (or both). Give an efficient algorithm that, given a directed graph $G = (V, E)$, outputs whether or not it is semiconnected. Argue the correctness of your algorithm and analyze its time complexity to be $\Theta(V + E)$.

5. **All Pairs Shortest Paths** (10 + 5 + 5 + 5 + 5 + 5 = 35 points)

   You saw the power of dynamic programming when designing the Bellman-Ford algorithm. You will see its power once more when solving this problem.

   Let G be a digraph, with edge weights given by $w : E(G) \to \mathbb{R}$. Assume that $V(G) = \{1, 2, \ldots, n\}$. In class we study the Bellman-Ford algorithm that computes shortest paths to all vertices from a given source vertex $s \in V(G)$. Suppose instead that we wish to compute the shortest paths from every vertex to every vertex, i.e., we want to compute $\delta(i, j)$ for all $i, j \in V(G)$, where $\delta(i, j)$ is the weight of a shortest path from i to j. One solution would be to run Bellman-Ford n times, once for each $s \in V(G)$, but this solution would take $V \times O(nm) = O(n^2 m)$ time. Since m can be as high as n^2, the solution can be as expensive as $O(n^4)$. Can we do better? The answer is yes: we can design a $O(n^3)$-time algorithm. How? Well, read on.

   If $i_1, i_2, \ldots, i_k$ is a path p in a graph G, we call $i_2, i_3, \ldots, i_{k-1}$, which are all but the start and end vertices on path p, the *intermediate vertices* on p. For example, if p is 3, 6, 2, 1, 4, the intermediate vertices are 6, 2, 1. For another example, if p is 3, 6, 2, 4, 1, 4, then the intermediate vertices are 6, 2, 4, 1 (notice that even though 4 is the end vertex, it is counted as an intermediate vertex since it also occurs as a non-start and a non-end vertex).

   Let $P^k_{i,j}$ denote the set of all paths p such that p is a path from vertex i to vertex j and every intermediate vertex on p is at most k. Define $d^k_{i,j}$ to be the minimum weight of a path in $P^k_{i,j}$. More precisely:

   $$d^k_{i,j} := \begin{cases} \infty & \text{if } P^k_{i,j} = \emptyset \\ -\infty & \text{if a path in } P^k_{i,j} \text{ contains a negative weight cycle} \\ \min\{w(p) \mid p \in P^k_{i,j}\} & \text{otherwise.} \end{cases}$$

   Now answer the following.

   (a) Write a recurrence for $d^k_{i,j}$, including base cases(s).

   (b) Use this recurrence to write an algorithm that, given an adjacency matrix A for a weighted digraph G, returns the following two quantities, for all $i, j \in V(G)$:

      i. $\delta(i, j)$

      ii. $\pi(i, j)$, where $\pi(i, j)$ is the last vertex before j on a shortest path from i to j if $i \neq j$ and $\delta(i, j) \notin \{-\infty, \infty\}$; and $\pi(i, j) = nil$ otherwise.

Your algorithm should run in $O(n^3)$ time and take $O(n^3)$ space.

You can assume that $A[i,i] = 0$ if $(i,i) \notin E(G)$ and $A[i,i] = w(i,i)$ otherwise; and, if $i \neq j$, then $A[i,j] = \infty$ if $(i,j) \notin E(G)$ and $A[i,j] = w(i,j)$ otherwise.

You don't need to argue the correctness or of your algorithm or its time or space complexity, but it should be of course correct, run in $O(n^3)$ time and take $O(n^3)$ space.

(c) State how to modify the above algorithm so that it still runs in $O(n^3)$ time, but takes only $O(n^2)$ space. Your algorithm should of course remain correct, run in $O(n^3)$ time and take $O(n^2)$ space, but you don't need to prove any of these facts.

(d) Write the invariant for only the outermost loop in your algorithm for Part (c). You don't need to prove your invariant correct.

(e) Write a method that, given $i, j \in V(G)$, uses the information computed by your algorithm to output a shortest path from i to j if $\delta(i,j) \notin \{-\infty, \infty\}$.

(f) Write a method that outputs whether G has a negative weight simple cycle and, if it does, it outputs a simple cycle in G. Your method is not required to do anything more than what I asked for (for instance, it can stop and return as soon as it is able to answer the question asked).

(Recall that a simple cycle is a cycle where no vertex, other than the first and last, occurs more than once.)