Compsci 330 Design and Analysis of Algorithms
Assignment 5, Spring 2024 Duke University

Example Solution

Due Date: Thursday, February 22, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

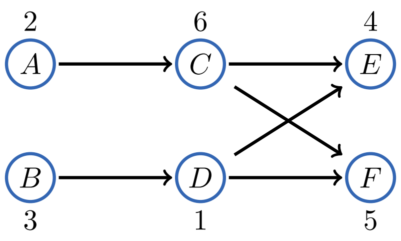
Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (DAG Reachability). You are given a directed acyclic graph $G = (V, E)$ with n vertices and m edges, in which each node $u \in V$ has an associated *price* p_u which is a positive integer. Define the array COST as follows: For each $u \in V$,

$$\textsc{cost}[u] = \text{price of the cheapest node reachable from } u \text{ (including } u \text{ itself).}$$

For instance, in the graph below (with prices shown for each vertex), the COST values of the nodes A, B, C, D, E, F are $2, 1, 4, 1, 4, 5$, respectively.



Describe an algorithm that completes the *entire* COST array (i.e., for all vertices) in $O(m+n)$ time. Briefly explain the correctness of the algorithm and analyze its runtime complexity.

You may use depth-first search (DFS) or Kosaraju's SCC algorithm as described in lecture without restating the algorithm or arguing for its correctness.

Solution 1. The algorithm can be described in four major steps.

(i) Perform DFS on G, store u.post for each vertex u. The details of this can be seen in class or Erickson 6.1.

(ii) Store the vertices u in an array V_R sorted in *increasing* order of u.post. Note that this is the reverse of the typical topological order of G.

(iii) Traverse the array V_R sequentially (in reverse topological order, intuitively from sinks to sources).

(iv) For each vertex u: initialize $\textsc{cost}[u]$ by p_u. Iterate over all of the vertices reachable by one edge from u, that is: $\{v : u \to v \in E\}$. If $\textsc{cost}[v] < \textsc{cost}[u]$, update $\textsc{cost}[u]$ to $\textsc{cost}[v]$.

Correctness. For a vertex u_i, let $D[u_i]$ be the set of vertices reachable from u_i. Since any path from u_i to a vertex $u_j \neq u_i$ in $D[u_i]$ has to pass through an out-neighbor u_k of u_i, and every vertex reachable from u_k is also reachable from u_i, we have

$$D[u_i] = \{u_i\} \cup \left(\bigcup_{u_k : u_i \to u_k \in E} D[u_k] \right).$$

By definition,

$$\textsc{cost}[u_i] = \min_{v \in D[u_i]} p_v = \min\{p_{u_i}, \min_{u_k : u_i \to u_k \in E} \textsc{cost}[u_k]\}.$$

Since the vertices are stored in V_R in increasing order of their completion time, if $u_i \to u_k$ is an edge of E, then $k < i$. Therefore, all out-neighbors of a vertex u_i are stored before u_i in V_R.

Runtime. Running DFS on G takes $O(m+n)$ time. The procedure that follows visit each vertex and edge exactly once, which also takes $O(m+n)$ time. Therefore, the total running time is $O(m+n)$.

Problem 2 (Dining Preferences). Suppose m friends want to eat dinner together and there are n possible restaurants to choose from. Each friend k reports an array A_k of length $n_k \leq n$ where each element is a restaurant, and A_k is sorted in decreasing order of preferred restaurants for the friend k. Set $N = \sum_{i=1}^{k} n_i$. Describe an $O(n + N)$-time algorithm that either computes an array B of length n sorted in a way that is consistent with the preference of all of the friends or reports that none exists. That is, for all $1 \leq k \leq m$ and $1 \leq j < n_k$, $A_k[j]$ appears before $A_k[j+1]$ in B, or reports that no such ordering exists.

For example, suppose there are three restaurants and three friends, and their preferences are $A_1 = [a, c], A_2 = [c, b]$ and $A_3 = [a, b]$. Then (a, c, b) is a valid ordering of the three restaurants.

Briefly explain the correctness of the algorithm and analyze its runtime complexity.

You may use depth-first search (DFS) or Kosaraju's SCC algorithm as described in lecture without restating the algorithm or arguing for its correctness.

Solution 2. Label the restaurants $1, \dots, n$. We construct a directed graph $G = (V, E)$, as follows.

- $V = \{1, \dots, n\}$, that is, there is a node for every restaurant.
- For each friend k and for each $1 \leq j < n_k$, we add the directed edge $A[j] \to A[j+1]$ to G.

Next, we perform a depth-first search (DFS) on G, checking for back edges and keeping track of recursive post-times as described in class.

- If there is a back edge, report that no such B exists.
- Otherwise, sort the restaurants labeled $1, \dots, n$ in decreasing order of their post times from the and return this order for B. In other words, topologically sort the vertices node in v.

Correctness. We consider the following two cases:

- Case 1: We detect a cycle and return that no such B exists. Suppose for a contradiction there is such a B. Take the subsequence of restaurants in B that constitute the cycle $B[c_1], B[c_2], ..., B[c_p]$ where $c_1 < c_2 < \cdots < c_p$ are the indices of the restaurants in the cycle. Since there is a cycle on these vertices, there must exist an edge of the form $B[c_j] \to B[c_{j'}]$ for $c_{j'} < c_j$, implying some friend prefers c_j over $c_{j'}$. This contradicts the validity of B.
- Case 2: There is no cycle in the graph and we return the topological order of the vertices. Such an order has the property that there is no edge from an earlier to a later vertex, which is precisely the desired property for the order of restaurants in B.

Runtime. Construction of G takes $O(n + N)$ time, and it has at most N edges. Therefore DFS on G takes $O(n + N)$ time, and we spend another $O(n)$ time to report the vertices of G. The overall running time of the algorithm is $O(n + N)$.

Problem 3 (Transit Connections). The NotUnited airline company operates in locations numbered $1, 2, \ldots, n$ and is based in location 1. NotUnited offers m connections between these locations. These connections are specified by a list of tuples where (i, j) means that there is a connection from location i to j. Connections are one way.

NotUnited guarantees that it is possible to reach all $n - 1$ additional operating locations via connections (possibly more than one) originating at their base in location 1. However, some customers have complained about finding themselves unable to return home via NotUnited connections after traveling to another destination. Describe an $O(n+m)$ runtime algorithm to compute the minimum number of connections that the company would need to add in order to ensure that this cannot happen. Briefly explain the correctness of the algorithm and analyze its runtime complexity.

You may use depth-first search (DFS) or Kosaraju's SCC algorithm as described in lecture without restating the algorithm or arguing for its correctness.

Solution 3. The algorithm can be described in the following stages.

(i) Define a graph $G = (V, E)$ where there is a vertex for every location $1, 2, \ldots, n$ and an edge (i, j) for every connection of that form.

(ii) Use Kosaraju's strongly connected components (SCCs) algorithm as described in lecture to compute the strongly connected components of G.

(iii) From the SCCs, compute the condensation G' of G with a single node for each SCC an an edge from SCC i to j if and only if there exists some edge from i to j.

(iv) Count the number of sink nodes in G'. Return this value.

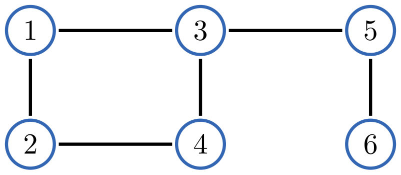
Correctness. We assume the correctness of the algorithm for computing the SCCs and therefore the condensation G' of G. Note that the resulting G' is a DAG. Since we are guaranteed that all locations can be reached from location 1, we know that there must be a SCC containing location 1 from which every other SCC in G' can be reached. Suppose there are x sink nodes in G'. We claim that x is the minimum number of connections that must be added in order to ensure that there is always a path from any node to any other. Consider adding a connection any node in every sink SCC to location 1.

First we argue that x is sufficient. After this modification, consider traveling from an arbitrary u to another arbitrary location v. There must be a path from u to a sink SCC in G', and since an SCC is strongly connected, to whichever node to which we added a connection to location 1. We are then guaranteed that there is a path from 1 to v. So there is a path from u to v.

Now we argue that x is necessary. Using fewer than x new connections, there will be some sink SCC in G' to which no connection is added. Since this SCC is a sink in G', any node in this SCC will not be able to reach any node not in this SCC.

Runtime. Constructing an adjacency list representation of the graph takes $O(n + m)$ time, and the resulting graph has n vertices and m edges. Kosaraju's SCC algorithm as discussed in class can be implemented in $O(n + m)$ time. Constructing the condensation also takes linear time, as we merely loop over the original graph adjacency list and add connections between components as they appear. G' necessarily has at most n nodes and m edges. Finally, we can count the number of sinks in linear time by looping over G' and, for each node, checking if it has an empty adjacency list. The overall runtime is therefore $O(n + m)$.

Problem 4 (Applied). You are given a computer network represented as a connected undirected graph $G = (V, E)$ where computers and routers are the vertices and connections between them are the edges. Say that a node in this network is a *critical point* if removing the node (and its incident edges) increases the number of connected components in the network (for example, if it disconnects the network). For example, in the following diagram, 3 and 5 are both critical points.



You will need to **design and implement** an algorithm that returns a list of all the critical points (as ints) in **any order** in $O(n + m)$ time (where $n = |V|$ and $m = |E|$).

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **Python.** You should submit a file called `critical.py` to the Gradescope item "Homework 5 - Applied (Python)." The file should define (at least) a top level function `critical` that takes in two inputs, an int n, which defines the number of vertices, and a list of tuples `(i:int, j:int) edges` where `(i,j)` refers to an undirected edge from node `i` to node `j` and return a list of `int` where each element is one of the critical points of the graph (in any order). The function header is as follows
  - `def critical(n:int, edges:list[(int, int)]):`
- **Java.** You should submit a file called `Critical.java` to the Gradescope item "Homework 5 - Applied (Java)" The file should define (at least) a top level function `critical` that takes in one 2D int array: `edges` and returns an ArrayList<Integer> where each element is one of the critical points of the graph (in any order). The header for the method is as follows
  - `public ArrayList<Integer> critical(int n, int[][] edges);`

  edges is a k by 2 2D arrays, where edges[i] is the undirected edge from edges[i][0] to edges[i][1]

Hint.. To begin, consider the depth-first search tree of the graph. For the root of the search tree, how can you tell if it is a critical point? For the other nodes, how can you tell by looking at the back edges of the tree? Finally, how can you use the pre-visit times to solve the problem in linear time?

Solution 4. As the hint suggests, we can determine the critical points using depth-first search (DFS) and the associated properties of the DFS tree. The key observation is that a node is a critical point if its removal causes a disconnection. If the root of the DFS tree has more than one child, then it is a critical point. This is because, by removing the root, the DFS tree becomes disconnected. For any non-root node v, let T_v be the subtree rooted at v. If there is a child u of v such that there is no back edge from T_u to the ancestors of v, then v is a critical point. We can determine the above properties using the pre-visit and post-visit times from the DFS.

```
procedure FindCriticalPoints($G(V, E)$)
    Perform DFS on $G$ to compute the DFS tree and pre-visit times.
    Initialize an array $low$ of size $|V|$ to $\infty$.
    for each vertex $v$ in reverse post-order do
        for each child $u$ of $v$ do
            $low[v] = \min(low[v], low[u])$          ▷ Update low values using child nodes
        for each back edge $(v, w)$ from $v$ do
            $low[v] = \min(low[v], \mathrm{pre}[w])$          ▷ Use back edges to update low values
    Let $v$ be the root of the DFS tree.
    if number of children of $v > 1$ then
        Mark $v$ as a critical point.
    for each vertex $v$ except the root do
        if there exists a child $u$ of $v$ such that $low[u] \geq \mathrm{pre}[v]$ then
            Mark $v$ as a critical point.
```

Correctness. If the root of the DFS tree has more than one child, then removing the root will cause the graph to break into multiple disconnected subgraphs, as none of the children of the root can reach the others without going through the root. Thus, the root is a critical point if it has more than one child. For non-root nodes, the algorithm employs the idea of low-link values. The low-link value of a node v is the smallest pre-order number of any node reached from v via the DFS tree, possibly using one back edge. If a vertex v has a child u such that none of the vertices in the subtree rooted at u has a back edge to a predecessor of v, then v is an articulation point (or critical point). This condition translates to

$$\mathrm{low}[u] \geq \mathrm{pre}[v]$$

The algorithm first calculates the low-link values for each vertex using both tree edges and back edges. Then, using these values, it determines which vertices are critical points.

Runtime.

- *DFS Traversal:* The DFS traversal of the graph, where vertices represent computers/routers and edges represent connections, will take $O(n + m)$ time, where n is the number of vertices and m is the number of edges.
- *Low-Link Value Computation:* For each vertex, the algorithm computes the low-link value, which involves iterating over all children of the vertex and all back edges from the vertex. Since each edge is considered at most twice (once for each of its two vertices), this computation takes $O(n + m)$ time.
- *Critical Point Determination:* The algorithm then determines which vertices are critical points by examining each vertex's children (or for the root, whether it has more than one child). This process is linear in the size of the graph, so it takes $O(n + m)$ time.

Therefore, the algorithm has a runtime complexity of $O(n + m)$.