Compsci 330 Design and Analysis of Algorithms
Assignment 2, Fall 2024 Duke University

Example Solution

Due Date: Monday, September 16, 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.

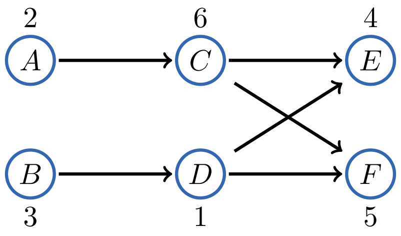
Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[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. Recall that a node v is *reachable from* node u if there is a path from u to v. Define the array COST as follows: For each $u \in V$,

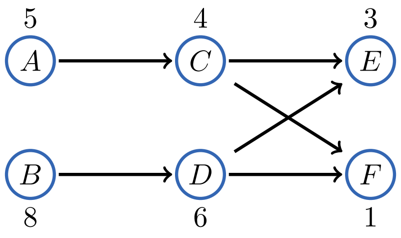
$$\textsc{cost}[u] = \text{price of the cheapest node reachable from } u \text{ (including } u \text{ itself).}$$

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.

Updated 9/13: The following example graph has COST value 1 for vertices A, B, C, D, F and value 3 for E. Note that node F is reachable from every node except E.



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

(i) Store the vertices u in an array V_R sorted in *reverse* topological order of u.

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

(iii) 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. From lecture, it is given that a topological ordering can be computed in $O(m + n)$ since G is a DAG. 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 (2SAT). In the 2SAT problem, you are given a set of *clauses*, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value True or False to each of the variables so that *all* clauses are satisfied—that is, there is at least one true literal in each clause. For example, consider the following instance of 2SAT:

$$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_4).$$

This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to True, False, False, and True, respectively. For practice (do not submit), find another satisfying assignment for this instance, then come up with another instance with four variables for which there is no satisfying assignment.

For this problem you will describe an efficient algorithm to solve 2SAT efficiently by reducing it to the problem of finding the strongly connected components (SCCs) of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$, (where α, β are literals), G_I has an edge from the negation of α to β, and one from the negation of β to α. For example, if the clause is $(\overline{x}_1 \vee x_2)$, then there is an edge from $\overline{\overline{x}}_1 = x_1$ to x_2 and an edge from $\overline{x}_2$ to $\overline{x}_1$.

Note that the clause $(\alpha \vee \beta)$ is logically equivalent to the implications $\overline{\alpha} \Rightarrow \beta$ and $\overline{\beta} \Rightarrow \alpha$. In this sense, G_I records all of the clauses in I by representing their equivalent implications as edges in G_I. For practice, consider drawing G_I for the example instance above.

(a) Show that if G_I has a SCC containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

(b) Next show the converse of (a): namely, that if none of G_I's SCCs contain both a literal and its negation, then the instance I must be satisfiable. *[Hint: Assign values to the variables as follows: repeatedly pick a sink SCC (i.e., an SCC without edges leaving the SCC). Assign value True to all literals in the sink, assign False to their negations, then delete these (the literals and their negations) from G_I. Show this results in a satisfying assignment.]*

(c) Conclude that there is a $O(m + n)$-time algorithm for 2SAT.

Solution 2.

(a) For sake of contradiction, suppose G_I has a strongly-connected component (SCC) containing both x and $\overline{x}$ for some variable x and I has a satisfying assignment; that is, there is an assignment of True/False values so that all clauses of I are satisfied. Suppose that the literals of I are assigned True/False so that all clauses are satisfied. By definition of SCC, there is a path from x to $\overline{x}$ in G_I and a path from $\overline{x}$ to x. Exactly one of $x, \overline{x}$ must be True and the other False in any valid assignment, including the assumed satisfying assignment. Thus one of these paths begins at a True vertex and ends at a False vertex. Without loss of generality, suppose x is True, so $\overline{x}$ is False and the path of interest is from x to $\overline{x}$. Since this path begins with True and ends with False, there must be an edge $u \to v$ on this path where u is True and v is False (at some point along the path, it must switch from True vertices to a False vertex). By construction of G_I, this edge $u \to v$ exists in the graph because $(\overline{u} \vee v)$ is a clause of the instance I. Since u is True and v is false, this clause evaluates to False. This contradicts the assumption that all clauses are satisfied, which completes the proof.

(b) The hint gives the algorithm to use. There are three main things to prove: (i) the negations of all literals in the sink SCC lie in a source SCC, (ii) by assigning True to all literals in the sink (and thus False to all literals in the corresponding source) no edges in the DAG are of the form True to False, and (iii) By never introducing a True to False edge in the process, the overall algorithm terminates with a satisfying assignment.

Claim (i) follows from the fact that if $\alpha \to \beta$ is an edge of the DAG, then $\overline{\alpha} \vee \beta$ is a clause in I, and thus $\overline{\beta} \to \overline{\alpha}$ is also an edge of the DAG. The literals of the latter are the negations of the literals in the original edge. Thus, for all edges in the sink SCC, there is a reverse edge between the negations of the literals. So the negations are strongly-connected, which implies the SCC containing the negations consists of only the negations. Furthermore, since the sink SCC is a sink, it has no outgoing edges in the DAG, so there are no incoming edges into the SCC with the negations; that is, it is a source SCC.

To see claim (ii), suppose there is an edge $\alpha \to \beta$ where α is assigned True and β is assigned False during the algorithm. When True was assigned to α, it was contained in a sink SCC, and when False was assigned to β, it was in a (different) source SCC. For the SCC containing α to be a sink and subsequently deleted, the SCC containing β must have been previously deleted. However, for the SCC containing β to be a source and subsequently deleted, the SCC containing α must have been previously deleted. The SCCs cannot be both deleted before each other, so this cannot happen. We conclude there are no edges of the form True to False.

To see claim (iii), all edges are of the form True to True, False to True, or False to False. All three kinds of edges correspond to satisfied clauses. All clauses correspond to an edge (in fact two edges) of this form, so all clauses in I are satisfied by the assignment.

(c) The algorithm involves constructing the SCC DAG, which takes $O(n + m)$ time. In order to efficiently pick and remove a sink (and source SCC) from the DAG, we first compute a topological ordering of the DAG in $O(n' + m') = O(n + m)$ time, where $n' \leq n$ and $m' \leq m$ are the number of vertices and edges in the DAG, respectively. By iterating through the vertices of the DAG in reverse topological order, we find a next sink SCC of the DAG to consider and remove, along with its corresponding source SCC. When we pick a sink SCC to remove, we spend constant time for every literal in that SCC and their negations, then remove the sink and corresponding source from the DAG in constant time. Each literal appears once in the DAG. Thus, summing over all SCCs, the total time spent is $O(n + m)$ to pick and delete the sink/source SCCs while iterating through the DAG. Constructing the DAG and its topological ordering takes an additional $O(n + m)$ time. Thus the total runtime is $O(n + m)$.

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 wishes to add connections so that it is possible to reach all $n-1$ additional operating locations via a sequence of connections (possibly only one) starting from their base at location 1. Describe an $O(n+m)$ runtime algorithm to compute the minimum number of connections that the company would need to add. If no new connections are necessary, return zero. Briefly explain the correctness of the algorithm and analyze its runtime complexity.

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) Compute the strongly connected components of G.

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

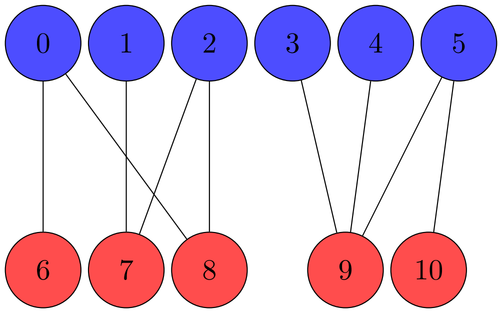
(iv) Count the number k of source nodes in G'. If location 1 is not in such a source node then return k, and otherwise return $k-1$.

Correctness. First, note that every source SCC of the condensed DAG (except possibly the one containing location 1) is not reachable from location 1. Thus, at minimum, we require one edge from location 1 into every source SCC (possibly one less if 1 is in a source SCC). It actually turns out that these new connections are sufficient! To prove this, we claim that all locations are reachable from a source SCC. Consider any non-source SCC. Since it is not a source, there is some incoming edge from a neighboring SCC. We repeat this process with this neighboring SCC, essentially going backwards through the DAG, until we eventually end up at a source. This source by our initial construction will be reachable from 1, so our original non-source SCC must also be reachable from 1.

Runtime. Constructing an adjacency list representation of the graph takes $O(n+m)$ time, and the resulting graph has n vertices and m edges. As discussed in class, steps (ii) and (iii) can be implemented in $O(n+m)$ time. G' necessarily has at most n nodes and m edges. Finally, we can count the number of sources 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). Bipartite graphs are a special class of undirected graphs of the form $G = (V, E)$ that have a *2-coloring*, which is an assignment of either red or blue to each vertex such that every edge has one red endpoint and one blue endpoint. Some problems can be solved faster on bipartite graphs than general graphs. Thus, it is important to be able to determine when a given graph is bipartite so that we can safely run algorithms tailored for bipartite graphs when appropriate.

For this problem, you are given an undirected graph $G = (V, E)$ whose vertices are valued 0 to $n - 1$, where $n = |V|$. G may or may not be connected. We call a 2-coloring of a bipartite graph as *least-red* if it colors the fewest possible vertices as red (and the rest blue). Your task is to determine whether G is bipartite, and if so, return a **least-red** 2-coloring represented by a Boolean array $R[1..n]$, where $R[i]$ is True if vertex i is red and False if vertex i is blue, for each i from 0 to $n - 1$.



In the example bipartite graph above, one of two distinct least-red 2-colorings are shown: vertices $6, 7, 8, 9, 10$ are red and vertices $0, 1, 2, 3, 4, 5$ are blue.

You will need to **design and implement** an algorithm that, given $G = (V, E)$, either reports no 2-coloring exists (represented by returning `None` in Python or `null` in Java), or a least-red 2-coloring in $O(n + m)$ time (where $n = |V|$ and $m = |E|$). G may have multiple least-red 2-colorings (how?), in which case you may output any one of them.

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.

- **Python.** You should submit a file called `least_red.py` to the Gradescope item "Homework 2 - Applied (Python)." The file should define (at least) a top-level function `least_red` 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`, that returns `None` if no 2-coloring of the graph exists, and otherwise returns a list of n `bool`s that represents a least-red coloring, where each i-th value is `True` if and only if vertex i is red in the coloring. The function header is as follows
  - `def least_red(n:int, edges:list[(int, int)]):  -> list[bool]`
- **Java.** You should submit a file called `LeastRed.java` to the Gradescope item "Homework 2 - Applied (Java)" The file should define (at least) a top level function `leastRed` that takes in one 2D int array, `edges`, and either returns `null` if no 2-coloring exists, or otherwise returns an `ArrayList<Boolean>` that represents a least-red coloring, where each i-th value is `True` if and only if vertex i is red in the coloring. The header for the method is as follows

– `public ArrayList<Boolean> leastRed(int n, int[][] edges);`

`edges` is a $m \times 2$ 2D int array, where `edges[i]` is the undirected edge from `edges[i][0]` to `edges[i][1]`.

Solution 4. The main idea is to use BFS to detect whether the graph is bipartite. This amounts to checking if each connected component of the graph is bipartite.

We use BFS from a yet-unvisited/colored vertex, then try to color it and all other vertices in its (to be explored) connected component with two colors. Initially, we color this vertex, say, blue. As we perform BFS to explore the edges of its connected component, we try to color their endpoints so that one is red and one is blue. One endpoint is always already visited and thus already colored, and the other may or may not be colored already. If we find an edge whose endpoints have already been colored, but this edge requires us to recolor an endpoint differently than before, this connected component, and thus the graph, is not bipartite, and we return a value to represent this. Otherwise, we succeed at coloring all of the connected component with two colors, and it remains to check whether the 2-coloring is a least-red 2-coloring of the component. To do so, we check how many vertices are blue and how many are red; if there are more red than blue, we swap all vertices' colors in this component. (Note that these two colorings are the only valid 2-colorings of a component.)

We repeat this process until all connected components have been visited and colored. The final coloring of all nodes is composed of a least-red 2-coloring per component, and thus a least-red 2-coloring of the entire graph.