Computer Science CSC263H
St. George Campus

March 14, 2019
University of Toronto

Homework Assignment #6

**Due: March 28, 2019, by 5:30 pm**

- You must submit your assignment as a PDF file, named **a6.pdf**, of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  `https://markus.teach.cs.toronto.edu/csc263-2019-01`

  To work with one or two partners, you and your partner(s) must form a group on MarkUs.

- The **a6.pdf** PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTeX typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can use other typesetting systems if you prefer, but handwritten documents are not accepted.

- If this assignment is submitted by a group of two or three students, the **a6.pdf** PDF file that you submit should contain for each assignment question:

  1. The name(s) of the student(s) who *wrote* the solution to this question, and
  2. The name(s) of the student(s) who *read* this solution to verify its clarity and correctness.

- By virtue of submitting this assignment you (and your partners, if you have any) acknowledge that you are aware of the homework collaboration policy that is stated in the csc263 course web page: `http://www.cs.toronto.edu/~sam/teaching/263/#HomeworkCollaboration`.

- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.

- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.

- Your **a6.pdf** submission should be no more than 3.5 pages long in a 10pt font.

**Question 1.** (1 marks) Consider the *directed* graph $G$ with nodes $\{1, 2, 3, 4, 5, 6, 7\}$ that is represented by its adjacency lists:

$A(1) = 6, 4, 7$
$A(2) = 6$
$A(3) = 1, 7$
$A(4) = 5, 2, 7$
$A(5) = 2, 6$
$A(6) = \text{NIL}$
$A(7) = 6$

**a.** Draw a Depth-First Search forest generated by the DFS of $G$ under the following assumptions: *the DFS starts at node 1 and it explores the edges in the order of appearance in the above adjacency lists (e.g., edge* $(1, 6)$ *is explored before edge* $(1, 4)$*), and all the vertices are explored.* Do not draw forward, back, or cross edges. Show the discovery and finishing times $d[u]$ and $f[u]$ of every node $u$ of $G$, as computed by this DFS of $G$.

**b.** How many back edges, forward edges, and cross-edges are found by the above DFS?

**c.** Suppose the graph $G$ above represents seven courses and their prerequisites. For example, $A(1) = 6, 4, 7$ means that course 1 is a prerequisite for (i.e., it must be taken before) courses 6, 4 and 7; and $A(6) = \text{NIL}$ means that course 6 is not a prerequisite for any course.

Using Part (b) above and a theorem that we learned in class, prove that it is possible to take all the courses in a sequential order that satisfies all the prerequisite requirements. State the theorem that you use in your argument; do ***not*** give a specific course order here.

**d.** Now list the courses in an order they can be taken without violating any prerequisite. To do so you ***must*** use your DFS of Part (a) and an algorithm that we covered in a tutorial.

**e.** Draw a Breadth-First Search tree of $G$ that ***starts at node 3*** and explores the edges in the order of appearance in the above adjacency lists.

**Question 2.** (1 marks) In Question 2 of assignment 4 you solved a problem about "constraints", here you are asked to solve exactly the *same* problem but more efficiently.

You are given a list of $m$ *constraints* over $n$ distinct variables $x_1, x_2, ..., x_n$. Each constraint is of one of the following two types.

1. An *equality* constraint of the form $x_i = x_j$ for some $i, j$ where $1 \leq i \neq j \leq n$.
2. An *inequality* constraint of the form $x_i \neq x_j$ for some $i, j$ where $1 \leq i \neq j \leq n$.

For such a list of constraints, it may be possible to assign an integer to each variable such that this assignment does not violate any of the constraints in this list, or such assignment may not exist.

Design an efficient algorithm, which takes as input a list of $m$ constraints over $n$ variables, and outputs an assignment of integers to variables that satisfies all the constraints in this list, and if no such assignment exists the algorithm outputs NIL. Describe your algorithm in *clear and concise* English and analyze its worst-case time complexity. ***The worst-case running time of your algorithm must be*** $O(m + n)$***.***

HINT: Use a graph algorithm that we learned in class.

**Question 3.** (1 marks) Let $G = (V, E)$ be an undirected, connected graph, and $w : E \to \mathbb{R}$ be an edge weight function such that edges have *distinct* weights. Suppose that for every edge $e \in E$ there is a cycle of $G$ that contains $e$. Let $e_{max}$ be the edge with maximum weight in $G$. Prove that no minimum spanning tree of $G$ contains $e_{max}$.

[***The questions below will not be corrected/graded. They are given here as interesting problems that use material that you learned in class.***]

**Question 4.** (0 marks) Let $G$ be a digraph (i.e., a directed graph). A node $u$ of $G$ is called a *source* if there is no edge leading into $u$ (i.e., $u$ is not adjacent to any node).

Consider the following *source-deletion* operation, applied to a digraph that has at least one source: Arbitrarily choose a source $u$; remove from the graph the node $u$ and all edges incident from $u$.

Suppose we apply this operation repeatedly, until it can no longer be applied. There are two reasons why we may be unable to continue the process of applying the operation: either the remaining digraph has no sources, or it has no nodes at all!

**a.** Prove that if a non-empty digraph is acyclic, then it has at least one source.

**b.** Prove that a digraph is acyclic if and only if the repeated application of the source-deletion operation results in the empty graph. (Hint: Use part (a) for the only-if direction.)

**c.** Based on the characterisation of acyclicity shown in (b), you should develop an algorithm to determine whether a given digraph is acyclic, as described below. Your algorithm should run in $O(n+m)$ time, where $n$ is the number of nodes and $m$ is the number of edges of the input graph. You should assume the graph is input in the usual way, using adjacency lists.

Your algorithm should begin by computing the *in-degree* of each node. The in-degree of a node $v$ is defined to be the number of nodes $u$ such that $(u, v)$ is an edge. Your algorithm should compute, for each node $v$, the value $I(v)$ of the in-degree of $v$, as well as the set $S$ of nodes of in-degree 0. It should then, for each node in $S$ do the following: remove it from $S$, and effectively remove the node from the graph by changing the array $I$ appropriately, inserting new elements into $S$ as appropriate.

You should describe your algorithm first in clear English, and then also give the pseudo-code. Explain why your algorithm works, and why it has the stated time complexity.

**Question 5.** (0 marks) Suppose we have a road network between cities numbered from 1 to $n$. Each road connects a pair of cities, and can be traveled in both directions. The entire road network is represented by an undirected graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ are the cities, and the edges $E$ are the roads. Assume that the road network, i.e. the graph $G$, is connected. There are $k$ roads which have been damaged, and your goal is to choose which roads to repair, so that the there is a path between each pair of cities, and the total cost of repairs is minimized.

The roads $E$ are given to you in an array $R[1\,..\,m]$, where each array cell $R[i]$ contains two fields: $R[i].\mathit{edge}$ which contains the pair $(u, v)$ of cities that the $i$-th road connects, and $R[i].\mathit{cost}$ which contains the cost of repairing the road, if it is damaged, or 0 if it is not. The cost of repairing a damaged road can be assumed to be positive. Design an algorithm to compute a set of damaged roads to be repaired, so that, after the repairs, it is possible to travel between every pair of cities by following roads that are either undamaged or repaired. Moreover, the *total cost* of repaired roads should be minimized. Your algorithm should have worst-case running time at most $O(m \log^* m + k \log k)$. Describe your algorithm in clear and concise English, prove it is correct, and analyze its running time.

**Question 6.** (0 marks) Let $G$ be an arbitrary undirected connected graph where each edge has a weight, and $T$ be a minimum spanning tree of $G$.

**a.** Suppose we augment $G$ by adding a new node $v$ and new weighted edges connecting $v$ to some nodes of $G$. Let $G'$ be the resulting (weighted) graph. Prove or find a counterexample to the following statement: We can always obtain a minimum spanning tree $T'$ of $G'$ by adding to $T$ an edge of minimum weight among all the new edges (those connecting $v$ and the nodes of $G$).

**b.** Suppose we augment $G$ by adding a new edge $e$ of weight $w$ between two nodes of $G$. Let $G'$ be the resulting (weighted) graph. Describe an algorithm that builds a minimum spanning tree of $G'$ in $O(n)$-time, where $n$ is the number of nodes in $G$. Assume that each of $G$ and its minimum spanning tree $T$, is given in the adjacency-list representation. Your algorithm's english description should be clear and brief (do not use pseudo-code). Prove the correctness of your algorithm and explain why its worst-case running time is $O(n)$.