Computer Science CSC263H
St. George Campus

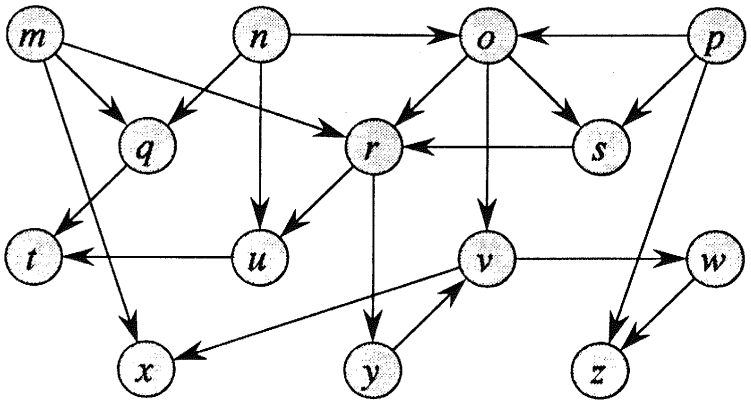
March 24, 2016
University of Toronto

Homework Assignment #6
Due: April 7, 2016, by 5:30 pm

- You must submit your assignment as a PDF file of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at `markus.cdf.toronto.edu/csc263-2016-01`. To work with a partner, you and your partner must form a group on MarkUs.
- The 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.
- By virtue of submitting this assignment you (and your partner, if you have one) acknowledge that you are aware of the policy on homework collaboration for this course.[a]
- 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.

[a] "In each homework assignment you may collaborate with at most one other student who is currently taking one of the sections of CSC263H taught this term. If you collaborate with another student on an assignment, you and your partner must submit only one copy of your solution, with both of your names. The solution will be graded in the usual way and both partners will receive the same mark. **Collaboration involving more than two students is not allowed. For help with your homework you may consult only the course instructors, teaching assistants, your homework partner (if you have one), your textbook and your class notes. You may not consult any other source.**"

Question 1. (30 marks) Consider the following graph G:



Note: the nodes of G in alphabetical order are m, n, o, p, q, r, s, t, u, v, w, x, y, z.

a. (10 marks) Draw the Breadth-First Search tree generated by executing the BFS algorithm on the above graph G under the following assumptions: (i) The BFS starts at node p *and* (ii) the graph G is given by adjacency lists that are in alphabetic order, so the BFS explores edges in lexicographic order whenever there is a choice (e.g., edge (p, o) is explored before edge (p, s)). Show the $d[a]$ value for every node a discovered by this BFS.

b. (10 marks) Draw the Depth-First Search forest generated by executing the DFS algorithm on the above graph G under the following assumptions: (i) the **for** loop of the DFS procedure considers the nodes in alphabetical order (so the DFS starts at node m) *and* (ii) the graph G is given by adjacency lists that are in alphabetic order, so the DFS explores edges in lexicographic order whenever there is a choice (e.g., edge (m, q) is explored before edge (m, x)). Show the $d[a]$ and $f[a]$ values for every node a of the graph G, as computed by this DFS.

c. (3 marks) Given an edge (a, b) in a graph and the corresponding $d[a]$, $d[b]$, $f[a]$, and $f[b]$ assigned to nodes a and b by some DFS of this graph, how do we know if this edge is a *back edge* with respect to this DFS? Justify your answer.

d. (3 marks) Prove that graph G has no cycles. To do so use your DFS of graph G and an appropriate theorem (which is given in the textbook and was shown in the course).

e. (4 marks) List the nodes of the graph G in the topological sort order *given by the DFS of G in part (b)* (when executing the TOPOLOGICAL-SORT algorithm described in the textbook and covered in a tutorial).

Question 2. (20 marks) We have n elements $\{1, \ldots, n\}$, and we are given as input a 2-dimensional array D of the pairwise distances between them: $D[i, j]$ is the distances between i and j. The distance of any element to itself is 0 (i.e. $D[i, i] = 0$), and, moreover, the distances are symmetric: $D[i, j] = D[j, i]$ for every i and j. We would like to partition the n elements into two disjoint sets S and $\bar{S}$ that are *as far apart as possible.*

More precisely, given a set $\emptyset \subset S \subset \{1, \ldots, n\}$, define the distance between S and its complement $\bar{S} = \{1, \ldots, n\} \setminus S$ as $d(S, \bar{S}) = \min_{i \in S, j \in \bar{S}} D[i, j]$, i.e. the *minimum* distance between a point in S and a point in $\bar{S}$. In this question your goal is to design an efficient algorithm to find a partition S and $\bar{S}$ such that the distance $d(S, \bar{S})$ between S and $\bar{S}$ is the *largest* among all the ways to partition the elements $\{1, \ldots, n\}$ into two sets. That is, your algorithm should find a partition S and $\bar{S}$ such that the following holds:

$$d(S, \bar{S}) = \max\{d(R, \bar{R}) : \emptyset \subset R \subset \{1, \ldots, n\}, \bar{R} = \{1, \ldots, n\} \setminus R\}.$$

Note that a "brute force" algorithm to do so would be very expensive. One way to do it is as follows:
(1) enumerate every possible partition S and $\bar{S}$ of $\{1, \ldots, n\}$,
(2) for each partition S and $\bar{S}$ find the smallest edge between S and $\bar{S}$ — this gives you $d(S, \bar{S})$, and
(3) find the partition S and $\bar{S}$ with the largest $d(S, \bar{S})$.

Since there are $\Theta(2^n)$ partitions of $\{1, \ldots, n\}$, this algorithm works in exponential time. Your algorithm should be much more efficient than this.

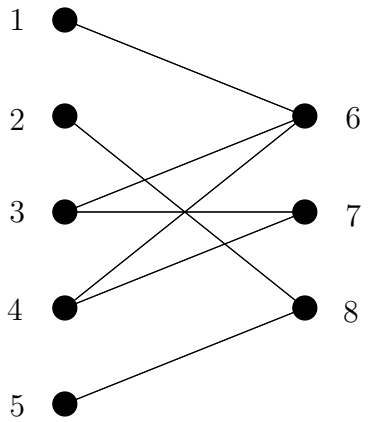
a. (10 marks) Let G be the complete graph with vertices $\{1, \ldots n\}$, with the weight of edge $e = (i, j)$ given by $w(e) = D[i, j]$. Let T be a minimum spanning tree of G end let $e = (i, j)$ be an edge of e. $T - e$ (the tree T with the edge e removed) has two connected components: let S be the vertices in one of them, and $\bar{S}$ the vertices in the other. Prove that $d(S, \bar{S}) = w(e) = D[i, j]$.

b. (8 marks) Use part (a) to derive an efficient algorithm to find a partition S and $\bar{S}$ such that the distance $d(S, \bar{S})$ between S and $\bar{S}$ is the largest among all the ways to partition the elements $\{1, \ldots, n\}$ into two sets, as described above. Describe your algorithm in clear and concise English, and prove that it is correct.

c. (2 marks) What is the worst-case time complexity of your algorithm? Justify your answer.

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

Question 3. (0 marks) Let $G = (V, E)$ be an undirected graph. G is ***bipartite*** if the set of nodes V can be partitioned into two subsets V_0 and V_1 (i.e., $V_0 \cup V_1 = V$), so that every edge in E connects a node in V_0 and a node in V_1. For example, the graph shown below is bipartite; this can be seen by taking V_0 to be the nodes on the left and V_1 to be the nodes on the right.



a. Prove that if G is bipartite then it has no simple cycle of odd length. Hint: Give a proof by contradiction.

b. Prove that if G has no simple cycle of odd length (i.e., every simple cycle of G has even length) then G is bipartite. (Hint: Suppose every simple cycle of G has even length. Perform a BFS starting at any node s. Assume for now that G is connected, so that the BFS reaches all nodes; you can remove this assumption later on. Use the distance of each node from s to partition the set of nodes into two sets, and prove that no edge connects two nodes placed in the same set.)

c. Describe an algorithm that takes as input an undirected graph $G = (V, E)$ in adjacency list form. If G is bipartite, then the algorithm returns a pair of sets of nodes (V_0, V_1) so that $V_0 \cup V_1 = V$, and every edge in E connects a node in V_0 and a node in V_1; if G is not bipartite, then the algorithm returns the string "not bipartite". Your algorithm should run in $O(n + m)$ time, where $n = |V|$ and $m = |E|$. Explain why your algorithm is correct and justify its running time. (Hint: Use parts (a) and (b).)

Question 4. (0 marks) In this question the goal is to find all the "peaks" in a topographical map. The input map is given as a 2-dimensional array H of size $n \times n$, where $H[i, j]$ is the height of the point (i, j). We say that a point (k, ℓ) is *adjacent* to a point (i, j) if $|i - k| \leq 1$ and $|j - \ell| \leq 1$. We say that a point q *can be reached* from a point p *by going down* if it is possible to start from p and eventually reach q by moving from one point to an adjacent point of the same or smaller height. (Note: staying on the same height also counts as going "down" according to this definition.)

The highest *peak* P_1 on the map is defined as follows: let $p_1 = (i, j)$ be the highest point on the map (i.e. $p_1 = (i, j)$ is such that $H[i, j] = \max\{H[i, j] : 1 \leq i, j \leq n\}$); then P_1 consists of p_1 and all points that can be reached from p_1 by only going down. The second highest peak P_2 on the map is defined as follows: let p_2 be the highest point *not contained in* P_1; P_2 consists of p_2 and all points *not in* P_1 that can be reached from p_2 by going down. In general, the i-th highest peak P_i is defined as: p_i is the highest point *not* in $P_1 \cup \ldots \cup P_{i-1}$, and P_i consists of p_i and all points that are not in $P_1 \cup \ldots \cup P_{i-1}$, and can be reached from p_i by going down.

Design an algorithm running in time $O(n^2 \log n)$ which lists the sizes of the peaks in order from the highest peak to the lowest. More precisely your algorithm should output $|P_1|, |P_2|, \ldots, |P_t|$, where t is the total number of peaks. Describe your algorithm in clear and concise English, explain why it is correct, and prove that its complexity is $O(n^2 \log n)$.