Homework 2 – Due Friday, September 5, 2014

Please refer to the general information handout for the full homework policy and options.

Reminders

- Your solutions are due before the lecture. Late homework will not be accepted.
- Collaboration is permitted, but you must write the solutions *by yourself without assistance*, and be ready to explain them orally to a member of the course staff if asked. You must also identify your collaborators. *Getting solutions from outside sources such as the Web or students not enrolled in the class is strictly forbidden.*
- To facilitate grading, please write down your solution to each problem on a separate sheet of paper. Make sure to include all identifying information and your collaborators on each sheet. Your solutions to different problems will be graded separately, possibly by different people, and returned to you independently of each other.
- For problems that require you to provide an algorithm, you must give *a precise description of the algorithm*, together with a *proof of correctness* and an *analysis of its running time.* You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

Exercises These should not be handed in, but the material they cover may appear on exams:

- Problems in KT, Chapters 1, 2 and 3.
- **(Graph representations)** Questions in this problem refer to the adjacency list and adjacency matrix representations defined on pages 87–89 of KT.

  You are given a directed graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges. Let G^R be the graph obtained by reversing the directions of all the edges in G. For each of the following, give an efficient algorithm and analyze its running time.

  If G is given in the adjacency list representation,

  1. compute the out-degree of each vertex;
  2. compute the in-degree of each vertex;
  3. compute the adjacency list representation of G^R.

  If G is given in the adjacency matrix representation,

  (d) compute the adjacency matrix of G^R.

- Give two algorithms to detect whether a given undirected graph has a cycle. If the graph contains a cycle, your algorithms should output one (not all of them, just one). Base the first algorithm on BFS and the second, on DFS. The running time of your algorithms should be the same as the running times of BFS and DFS. Explain why your algorithms are correct and run in the required time.
- A directed graph $G = (V, E)$ is **singly connected** if for all vertices u, v in V there is at most one simple path from u to v. (Recall that a path is simple if all vertices on the path are distinct.) Give an $O(mn)$-time algorithm to determine whether or not a directed graph is singly connected. Can you give an algorithm that runs in time $O(m + n)$?
- (**Shortest cycles containing a given edge**) Give an algorithm that takes as input a (undirected) graph $G = (V, E)$ and an edge $e_0 \in E$, and outputs a shortest cycle that contains e (if no cycle containing e exists, the algorithm should output "no cycle").

Problems to be handed in

1. **(Stable matchings)**
   (a) (Stable Matching with Indifferences) Chapter 1, problem 5.
   (b) (Truthfulness in Stable Matching, 1-page limit) Chapter 1, problem 8.
   Hint: Try playing with several specific examples of preference lists.
2. (**Asymptotics**)
   (a) Prove that for any positive numbers $a, b > 0$, we have $n^b = \omega(\log^a(n))$.
   There are several ways to approach this. One way is to use L'Hospital's rule for evaluating limits together with the following fact: $f(n) = \omega(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$.
   (b) Prove or disprove: There exist two *strictly increasing* functions $f(n), g(n)$, such that *neither* $f(n) = O(g(n))$ nor $g(n) = O(f(n))$ holds.
   (c) Prove or disprove: If $h(n) = \lceil n \log(n) \rceil$, then $n = \Theta(h(n)/\log h(n))$.
   (d) What is the asymptotic rate of growth of the number of paths of length k in a complete graph on n vertices?
3. * (**Optional problem. Electronic submissions only.**) The analysis of the Gale-Shapley algorithm establishes that every instance of the stable marriage problem admits at least one stable matching. Here we consider *how many* such matchings might exist.
   (NB: You must solve both parts of the problem to receive credit.)
   (a) Give an algorithm that takes an instance of the stable marriage problem as input and decides if there is *exactly one* stable matching for this instance (that is, the algorithm outputs either "unique stable matching", or "more than one stable matching"). Pay close attention to the proof of correctness of your algorithm.
   (b) Show that the maximum number of possible stable matchings for instances with n men and n women scales at least exponentially with n: that is, show that there is a constant $c > 1$, and a sequence of instances of the stable marriage problem, $x_1, x_2, ...$, one for each value of n, such that the number of stable matchings in instance x_n is at least c^n.