Homework 4

ECE 345 Algorithms and Data Structures
Fall Semester, 2017

Due: Friday, November 24 2017, 2PM

- All page numbers are from 2009 edition of Cormen, Leiserson, Rivest and Stein.
- For each algorithm you asked to design you should give a detailed *description* of the idea, proof of algorithm correctness, termination, analysis of time and space complexity. If not, your answer will be incomplete and you will miss credit. You are allowed to refer to pages in the textbook.
- Do not write C code! When asked to describe an algorithm give analytical pseudocode.
- Staple your homework properly. Use a stapler; do not use glue or other weird material to put it together. If you are missing pages, we are not responsible for it but you are!
- Write *clearly*, if we cannot understand what you write you may not get credit for the question. Be as formal as possible in your answers. Don't forget to include your name(s) and student number(s) on the front page!

---

1. **[Minimum Spanning Trees, 10 points]**

   The *bottleneck edge* in a path is the path's maximum-weight edge. A *minimax path* between two vertices s and t is a path such that no other path between s and t has a lighter bottleneck edge. Prove that any path in a minimum spanning tree of graph G is a minimax path of G.

2. **[Minimum Spanning Trees, 10+10+20 points]**

   (a) Assume that all edge costs in a graph G are distinct. Let C be a cycle in the graph, and let edge $e = (v, w) \in C$ be the most expensive edge of C. Prove that this edge cannot belong to any MST of G.

   (b) Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the minimum cost edge with one end in S and the other in $V \setminus S$. Prove that this edge belongs to every MST of G.

   (c) Use the previous two part to prove the following statement. Edge $e = (v, w)$ does not belong to a minimum spanning tree of graph G if and only if v and w can be joined by a path consisting entirely of edges that are cheaper than e. Make sure you prove both directions.

3. **[Graph Algorithms, 20 points]**

   When an adjacency matrix representation is used, most graph algorithms require time $O(n^2)$ (where n is the number of vertices), but there are some exceptions. Here's one.

Kumiko and her bandmates are having a election to determine the next school band president, and every band member is competing for it. Each candidate has a few preferences (people who the person would be willing to accept as as band president). Of course, the set of preferences for a person includes him/her self all the time.

What we are looking for is a "perfect" president who is in the set of preferences of every person and who does not prefer anyone but him/her self (wouldn't that make a good president?). In fact, all we want to know whether such a person exists or not. Otherwise, we're willing to live in anarchy. Define a directed graph with the set of candidates as the vertices and a directed edge from vertex a to vertex b if and only if b is in the set of preferences of a.

Suppose that the number of people (also the number of candidates) is n. Give an algorithm which executes in $O(n)$ time and determines if such a perfect presidential candidate exists or not. Assume that you are given the graph described above in the form of an $n \times n$ adjacency matrix.

4. **[Shortest Paths, 20+15 points]**

   (a) You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $u_0 \in V$. Describe an $O((|V| + |E|) \lg |V|)$ time algorithm for finding shortest paths between all pairs of nodes, with the restriction that these paths must all pass through u_0.

   (b) Consider a weighted directed graph $G = (V, E)$, where the edge weights are all positive integers, between 1 and c (a constant). Present an $O(|V| + |E|)$ time algorithm to solve the single-source shortest paths problem on this graph. (*Hint*: don't use Dijkstra. You may need to introduce pseudo-nodes to the graph.)

5. **[Topological Sort, 10 points]**

   Let $G = (V, E)$ be a **complete directed acyclic graph** that has an edge between every pair of vertices and whose vertices are labelled $1, 2, \ldots, n$, where $n = |V|$. To determine the direction of an edge between two vertices in V you are only allowed to ask a *query*. A query consists of two specified vertices u and v and is answered as follows:

   - "from u to v" if $(u, v) \in E$, or
   - "from v to u" if $(v, u) \in E$

   Give a worst-case lower bound (as a function of n) for the number of queries required to find a topological sort of G. (*Hint*: this problem can be solved by popular algorithms that we already know...)

6. **[Skewed Heaps, 20 points]**

   In this problem you are required to develop a data structure similar to that of the leftist heap from Homework 2. In *leftist heaps*, the `Merge` operation preserved the heap ordering and the balance (leftist bias) of the underlying tree. *Skewed heaps* use the same idea for merging heap-ordered trees. `SkewHeapMerge` is performed by merging the rightmost paths of two trees without keeping any explicit balance conditions. This means that there's no need to store the

rank of the nodes in the tree. This is similar to self-adjusting trees, since no information is kept or updated.

Good performance of those data structures is guaranteed by a "rebalancing step"—like the splay in self-adjusting trees, only simpler. At each step of the merging along the rightmost paths of the two heaps, we swap *all* of the left and right children of nodes along this path, except for the last one. The modified procedure for merging two skewed heaps looks as follows:

function SkewedHeapMerge(h,h') : heap

if h is empty then **return** h'
else if h' is empty then **return** h

if the root of $h' \preceq$ the root of h then
    exchange h and h' (* h holds smallest root *)

if right(h) = **nil** then
    right(h) := h' (* last node, we don't swap *)
else
    right(h) := SkewedHeapMerge(right(h), h')
    swap left and right children of h

return h

The above recursive routine can also be done iteratively. In fact, it can be done more efficiently than the leftist heap `Merge` (by a constant factor), because everything can be done in one pass, while moving down the rightmost path. In the case of leftist heaps, we go down the rightmost path, and then back up to recompute ranks. In leftist heaps, that also requires either a recursive algorithm or pointers from nodes to their parents [1].

Since there is no balance condition, there's no guarantee that these trees will have $O(\log n)$ worst-case performance for the merge (and hence all of the other operations). But they do have good *amortized performance*.

Here's the intuition for the above: In the previous merge algorithm we only had to swap children when the right one had a larger rank than the left. In this merge algorithm, we always swap children, so we might actually replace a right child of "small" rank with one of "large" rank. But the *next time* we come down this path, we will correct that error, because the right child will be swapped onto the left.

[1] Just a note on implementation here. Sometimes we may want to be able to move up a tree as easily as we move down; so every node will also include a pointer to its parent. That means that a node has a pointer to its left child, a pointer to its right child, and a pointer to its parent, increasing the amount of space needed to store trees. To decrease the *space* requirement, you can do this. Look at three nodes, a parent p and its two children r and l. Now p will have a pointer to l, but not to r; l has a pointer to r and r has a pointer back to p. So, all the left children in a tree have pointers to their right siblings (brothers) and to their own left children. All the right children have pointers to their parents and to their right children. If you draw a picture, this should make more sense. Now every node still has two pointers, and you can visit left nodes, right nodes and parent nodes easily.

(a) Show that a `SkewedHeapMerge` of two skewed heaps uses amortized time $O(\log_2 n)$ by the use of the *accounting method*.

(b) Show that `Insert` and `DeleteMin` have the same amortized time bounds.

(*Hint*: Use weights instead of ranks. Define the *weight* of a node to be the number of nodes in the subtree rooted at that node (below that node, including the node itself). Let node x be a *heavy* node if its weight is more than half of the weight of its parent. Otherwise it is *light* one. What can we say about light and heavy nodes in any binary tree? Look at any root–leaf path in the tree. *How many* light nodes can you encounter in such path? Why? The best case is when the light nodes are also right children of their parents. For the accounting method, every time we swap, we must pay \$1. Moreover, if it happens to swap a heavy child to a right child position, then you must deposit a "penalty" amount.)