CS 466/666, Fall 2007 due Friday Sept. 28

A. Lubiw

ASSIGNMENT 1

Do not copy from others, and acknowledge your sources.

1. [10 marks] Here are two review questions on NP-completeness. You may assume that the Hamiltonian Cycle Problem and the Travelling Salesman Problem are NP-complete.
   (i) Prove that the following problem is NP-complete: Given an edge weighted graph and a number K, is there a cycle that visits every vertex at least once (possibly more often) and has sum of edge weights at most K. NOTE: this differs from the standard TSP in that the cycle may re-visit some vertices.
   (ii) Prove that the decision version of TSP is *strongly NP-complete* meaning that it is still NP-complete if the input numbers are encoded in unary rather than binary.

2. [5 marks] Assuming that keys can be accessed ONLY by means of pair-wise comparisons, prove that for any implementation of the priority queue operations discussed in class (insert, delete-min, merge, decrease-key) the amortized cost of a sequence of k operations is $\Omega(\log k)$. Recall that amortized cost is an average over the sequence of operations, and a worst case with respect to the choice of the sequence.

3. [15 marks] Fibonacci heaps are complicated but have provably good amortized cost operations: $O(\log n)$ for delete-min and $O(1)$ for other operations, including the decrease-key operation. This good behaviour for decrease-key, is the justification for Fibonacci heaps, since this is the bottleneck in shortest path and min spanning tree algorithms. However, Fiboancci heaps do not behave as well in practice as a much simpler self-adjusting structure called the *pairing heap*. In a pairing heap, all values are stored in a single heap-ordered tree. No explicit shape constraints are imposed on the tree. The fundamental operation is *linking* two trees: the tree with the larger value at the root is added as a leftmost child of the other tree. A merge is simply a link. Insertion is implemented as a link of a singleton tree with the main tree. Decrease-key is implemented by cutting off the subtree rooted at that node, and linking the resulting two trees—the cut is done even if the decrease in key value would not violate heap order. Delete-min is implemented by deleting the root, which leaves a number of trees that must be joined back together somehow. Here's where the fun begins.
   (i) [2 marks] Show that all operations except delete-min take worst-case time $O(1)$. Show that delete-min can take worst-case time $\Theta(n)$ time no matter what joining rule is used.

   One simple joining rule is to use left-to-right incremental linking: $T \leftarrow T_1$; for $i = 2, \ldots k$ do $T \leftarrow link(T, T_i)$.
   (ii) [3 marks] Prove that delete-min implemented this way takes $\Theta(n)$ amortized time. Note that it does not suffice to show that a single delete-min takes $\Theta(n)$ time.

   The joining rule used for pairing heaps is slightly fancier and involves two passes. In the first pass, link the trees in pairs, T_1 with T_2, T_3 with T_4, etc. If k is odd then the last tree stays alone. In the second pass do right-to-left incremental linking.

(iii) [10 marks] Define the potential of a node with d children in an n-node heap to be $1 - \min\{d, [\sqrt{n}]\}$, and define the potential of a collection of heaps to be the sum of the potentials of their nodes. Using this potential function prove that insert, merge and decrease-key take $O(1)$ amortized time and that delete-min takes $O(\sqrt{n})$ amortized time.

Note (not needed to solve the problem). This proof will not use the exact ordering of the children in the two-pass join. There are orderings that can cause $\Omega(\sqrt{n})$ behaviour, but the particular ordering described above for pairing heaps results in an amortized cost of $O(\log n)$ for delete-min and for decrease-key. This upper bound does not seem to be tight for decrease-key, and in practice the behaviour of pairing heaps seems much better. An amortized lower bound of $\Omega(\log \log n)$ has been proved for the decrease-key operation.

Try to do this question without looking up the literature on pairing heaps. If you do resort to looking up the literature, be sure to acknowledge your sources.

4. [10 marks] There are sorting algorithms that beat the $\Omega(n \log n)$ lower bound if operations other than comparisons are allowed, in particular if operations on the bit representations of the numbers are allowed. You know radix sort, which can sort n integers in the range $[0, U]$ in linear time if U is n^k for k a constant (i.e. the integers have $O(\log n)$ bits). More recent results, such as fusion trees, beat the $O(n \log n)$ bound under weaker assumptions about U and n. The model of computing that is used allows *standard operations on words* at *unit cost per operation*. Standard operations include arithmetic, shifts, bit-wise AND, OR, etc. A word is w bits long, and it is assumed that: (1) each input number fits in one word; and (2) n fits in one word, which means that an index or pointer fits in one word.

(i) [2 marks] Does this seem like a reasonable model? Think about what happens if w is fixed, but also think about whether the model reflects reality.

At the lowest level, algorithms on this model depend on the ability to pack many small values in a single word so that one word operation essentially operates in parallel on all the values. Given integers $a_1, \ldots, a_t$, of b bits each, we can pack them into one word as follows. We use the least significant bits $0, 1, \ldots, b-1$ for a_1, then leave bit b as a spacing bit, then use the next b bits for a_2, etc. This way we can pack $t = \lfloor \frac{w}{(b+1)} \rfloor$ integers into a word of w bits. The i^{th} spacing bit occurs at bit position $ib + i - 1$.

(ii) [4 marks] Suppose that word A packs b-bit integers $a_1, \ldots, a_t$ as above, and word B packs b-bit integers $b_1, \ldots, b_t$. Suppose that all the spacing bits are initially set to 0. Show how to compute in constant time a word C such that the i^{th} spacing bit (from the right) gives the result of comparing a_i to b_i. Hint: step 1 is to construct a word that has 1's in the spacing bits and 0's elsewhere; start by assuming this.

(iii) [4 marks] Suppose that we want the result of comparing each a_i to one b-bit integer d. Show how to construct in constant time the appropriate word B so that the solution to part (ii) can be used.

Note. You might wonder how packing integers into one word of w bits is useful if the input integers have w bits, but the idea is to look at only some bits of the input integers at a time.