CSCI 570 Homework #4 Solution

Prof. Ming-Deh Huang
TAs: Iftikhar Burhanuddin, Chansook Lim
10/23/03

Donald E. Knuth (1938-) *Beware of bugs in the above code; I have only proved it correct, not tried it.*

Correction. 20-2(a) solution improved upon.

19.2-7

Proof Idea: The binary representation of $n = \langle b_{\lg n}, \ldots, b_0 \rangle_2$ which is the number of elements in the binomial heap H dictates the kind of binomial trees which can occur in H since B_i appears in H if and only if bit $b_i = 1$ (page 459, CLRS). Hence instead of thinking about binomial trees we can think of the 1's in the binary representation of n.

Incrementing a binary number is a special case of adding two binary numbers and so we'll only look at the latter and draw parallels with uniting binomial heaps.

The call to BINOMIAL-HEAP-MERGE in line 2 of BINOMIAL-HEAP-UNION is analogous to lining up two binary numbers which are to be added.

$$\begin{array}{cccc} & 1 & 0 & 1 \\ + & & 1 & 1 \\ \hline \end{array}$$

In the above example a binomial heap H_1 with $5 = \langle 101 \rangle_2$ elements is to be merged with a binomial heap H_2 with $7 = \langle 111 \rangle_2$ elements.

$$\begin{array}{cccccc} \text{carry} & 1 & 1 & 1 & & \\ \hline & & 1 & 0 & 1 & \\ + & & 1 & 1 & 1 & \\ \hline & 1 & 1 & 0 & 0 & \end{array}$$

In the above representation of binomial heap union there is a binomial tree B_0 in each heap which is merged to produce a B_1 tree denoted by the carry, which is inturn merged with the B_1 tree in H_2 to produce a B_2 tree. Now there are three B_2 trees (case 2 of BINOMIAL-HEAP-UNION), so one B_2 is left intact and the remaining two are merged to produce a B_3 tree and the union terminates.

The four cases that occur in BINOMIAL-HEAP-UNION are explained on page 466-467.

19.2-8

Proof Idea:

carry		1	1	
	1	0	1	1
+				1
	1	1	0	0

Observe that when inserting a node into a binomial heap essentially a B_0 (tree) heap is united with another heap. Case 2 doesn't occur and when case 1 occurs the union is complete.

Hence the BINOMIAL-HEAP-INSERT code is the BINOMIAL-HEAP-UNION code with lines $10-12$ replaced by the following

```
1.    do if (degree[x] ≠ degree[next-x])
2.       then return H
```

20.4-2

Proof Idea:

We are given that k is an integer constant. Let's assume that $k \geq 1$ as otherwise deleting k children doesn't make sense.

Let $G(i) = \min$ # of nodes in a subtree whose root has degree i and using analysis similar to lemma 20.1.

$$\begin{array}{rlcl} & G(i) & = & G(i-k) + G((i-1)-k) \ldots + G(0) + k \\ \Rightarrow & G(i+1) & = & G((i+1)-k) + G(((i+1)-1)-k) \ldots + G(0) + k \\ & & = & G(i+1-k) + G(i-k) \ldots + G(0) + k \\ & & = & G(i+1-k) + G(i) \\ \Rightarrow & G(i) & = & G(i-k) + G(i-1) \end{array}$$

The above recurrence with $G(0) = \ldots = G(k-1) = 1$ generates the Generalized Fibonacci sequence.

To solve this recurrence we make use of the characteristic polynomial method. The characteristic polynomial of the recurrence is obtained by shifting all terms to one side and replacing $G(i)$ by x^i which gives us $x^i - x^{i-k} - x^{i-1}$. Next factoring out the common term x^{i-k} we have $x^k - x^{k-1} - 1$.

Now a theorem (from the theory of solving recurrences in this way!) tells us that

$$G(i) = a_1 {\alpha_1}^i + \ldots + a_k {\alpha_k}^i \ (**)$$

where α_i are the roots of the characteristic polynomial and a_i are constants which can be computed based on the boundary (initial) values of the recurrence, which in our case are $G(0) = \ldots = G(k-1) = 1$.

By the Rule of Signs we see that the number of positive real roots of $f(x) = x^k - x^{k-1} - 1$ is the number of sign changes which is 1 in this

case $(+--)$ and the number of negative real roots of $f(x)$ is given by the number of sign changes of $f(-x) = (-x)^k - (-x)^{k-1} - 1$. When k is even $f(-x) = x^k + x^{k-1} - 1$ has 1 sign change $(++-)$ and so $f(x)$ has one real negative root. When k is odd $f(-x) = -x^k - x^{k-1} - 1$ has no sign change $(---)$ and so $f(x)$ has no real negative roots.

Now some calculus ...

$f'(x) = kx^{k-2}(x - \frac{k-1}{k}),\ f''(x) = k(k-1)x^{k-3}(x - \frac{k-2}{k})$

$f'(x) = 0 \Rightarrow x = 0, \frac{k-1}{k}.\ f''(\frac{k-1}{k}) < 0$. So when $x = \frac{k-1}{k},\ f(x)$ attains a (local) minimum and $f(\frac{k-1}{k}) < -1$.

$f(1) = -1,\ f(0) = -1$ so for $0 \leq x \leq 1,\ f(x)$ has the shape of a trough with (local) minimum attained at $x = \frac{k-1}{k}$.

When for $x > \frac{k-1}{k},\ f'(x) > 0$, that is slope of $f(x)$ is positive and hence $f(x)$ is an increasing function.

Since $f(1) = -1$ and $f(2) = 2^{k-1} - 1 > 0,\ f(x)$ will hit the x-axis (that is the real positive root will occur) somewhere in the interval $(1, 2)$.

The negative real root happens only in the even k case. Since $f(-1) = 1$ and $f(0) = -1,\ f(x)$ will hit the x-axis (that is the real negative root will occur) somewhere in the interval $(-1, 0)$.

Notice that in odd k case there is no negative real root and the positive real root is the only major contributor in $(**)$. In the even k case though a negative root occurs it is a fraction and high powers of it will not contribute in $(**)$. Hence in both cases we see that the positive real root α is the one that dominates.

In fact $G(i + k) \geq \alpha^i$ (which can be proved by mathematical induction. Basically boils down to proving $1 + \alpha^{k-1} = \alpha^k$ which is true since α is a root of $x^k - x^{k-1} - 1$. This is similar to 3.2-7, CLRS).

So $G(i) = G(i-1) + G(i-k) \geq \alpha^{i-1-k} + \alpha^{i-k-k} = \alpha^{i-2k}(\alpha^{k-1} + 1) \geq \alpha^{i-2k}\alpha^k = \alpha^{i-k}$.

We know that $n > G(i)$ so we have $\log_\alpha n > log_\alpha G(i) > \log_\alpha \alpha^{i-k} = i - k \Rightarrow (i-k) < \log_\alpha n \Rightarrow i < k + \log_\alpha n \Rightarrow i = O(\log n)$. As i (which denotes the degree) is bounded so is the maximum degree.

Observe that the previous inequality the logarithm in particular is defined provided $\alpha > 1$ which is always true when $k \geq 1$. Hence by choosing k to be constant and $k \geq 1$ (that is a node is cut from its parent as soon as it loses its kth child) we have $D(n) = O(\log n)$.

Sanity Check. Plugging in $k = 2$ gives us the $x^2 - x - 1$ the characteristic polynomial of the Fibonacci sequence the roots of which are $\phi, \hat{\phi} = \frac{1 \pm \sqrt{5}}{2}$ and here we have $\alpha = \frac{1+\sqrt{5}}{2}$ and it is the greater than 1 and the $G(i)$ equivalent is $F_i = \frac{1}{\sqrt{5}}\phi^i + \frac{1}{\sqrt{5}}\hat{\phi}^i$.

Remarks.

- The Generalized Fibonacci sequence.

  http://mathworld.wolfram.com/GeneralizedFibonacciNumber.html

- Solving recurrences using the characteristic polynomial refer to any book on discrete mathematics or combinatorics or

  http://mathcircle.berkeley.edu/BMC3/Bjorn1/Bjorn1.html

- René Descartes' Rule of Signs and its simple proof

  http://www.math.hmc.edu/funfacts/ffiles/20001.1.shtml

20-2

Proof Idea:

a.

The FIB-HEAP-CHANGE-KEY(H, x, k) operation in the case in which

- $k < key[x]$ will invoke the FIB-HEAP-DECREASE-KEY(H, x, k) operation which has an amortized cost of $O(1)$.
- $k = key[x]$ will just return after the comparison and hence has an amortized cost of $O(1)$ (due to the cost of comparison).
- $k > key[x]$ will invoke CUT$(H, y, p[y])$ for each child of x (i.e., $p[y] = x$) and then increase the key of x to k ($key[x] \leftarrow k$) and then invoke CUT$(H, x, p[x])$. Since the maximum degree of node in an n-node Fibonacci heap is $O(\log n)$ we have the amortized cost of the operation in this case is $O(\log n)$. (Alternatively we could $key[x] \leftarrow k$ and push x down the tree until there is no heap property violation by swapping x with a child with the minimum key at each level. The amortized cost is $O(\log n)$ if we maintain for each node a linked list for its children with a pointer to the child with a minimum key. This would incur an $O(1)$ extra cost for all other operations, but won't change the big-O cost.

b.

Deleting a node has an amortized cost of $O(\log n)$. If we were to delete r particular nodes the amortized cost would be $O(r \log n)$ but since the question says we could delete arbitrary nodes we hope to do better.

Deleting singleton trees and leaf nodes is easy. So by maintaining a pointer to leaf nodes in each tree, the amortized cost of pruning r nodes is r.

23.1-3

Proof Idea: Proof by contradiction aka reductio ad absurdum.

Let T be a MST for a graph G and the edge (u, v) be part of T. Since T is a tree, removing edge (u, v) disconnects the tree in two disjoint trees T_1 and T_2 (if T is still connected it would mean there was a cycle in tree T to begin with. A Contradiction!). Consider the particular cut which divides nodes in T_1 and T_2 into two different sets, i.e. the cut which respects nodes in T_1 and T_2. It is obvious that the cut is unique. Now, (u, v) must be one of the edges of the tree that are light edges crossing this cut, else, we could replace (u, v) with whatever is the light edge to obtain a spanning tree whose weight is less than T, which would lead to a contradiction. Hence given (u, v) is part of some MST, then it is a light edge crossing some cut of the graph.

23.1-5

Proof Idea: If e is not part of any MST for G then we are done as the claim holds trivially.

On the other hand suppose there is an MST T for G containing e. We will show that this leads to another MST T' not containing e.

Let T be a minimum spanning tree for G containing e. Remove e to obtain two disjoint subsets of nodes V_1 and V_2. Let e be incident on u and v. Since e is part of a cycle, there has to be a path $\mathcal{P}$ from u to v which doesn't involve e. Moreover, u and v belong to different subsets. Hence, there has to be an edge d in $\mathcal{P}$ crossing the cut (V_1, V_2). As e is a maximum-weight edge the weight of d is less than equal to the weight of e. (But as e is part of MST T, weight of d is equal to weight of e). Therefore, using d instead of e leads to the construction of MST T' for G' (not containing e) which is also a MST for G.

Note the importance of e being on some cycle of G. If e were not part of a cycle, not using e would lead to disconnection of the graph. And so e would have to be part of G's MST.

23.1-8

Proof Idea: Proof by contradiction using ideas in the previous problem.

Let T and T' be two different MSTs for the graph G such that the sorted lists for T and T', say L and L', are different. Let the first position of disagreement in the lists be k. Without loss of generality, assume $L[k] < L'[k]$.

Consider the edges corresponding to weight $L[1], L[2], \ldots, L[k]$, these edges form $n - k$ forests.

Now, we assert that there exists an edge whose weight is less than $L'[k]$ and adding it, results in a cycle with the maximum weighted edge in the cycle greater than equal to $L'[k]$.

The assertion follows from the following argument. Take the first $k-1$ edges in L'. The edges result in $n-k+1$ forests. Since the first k edges in L result in $n-k$ forests, there must be some edge amongst these k edges adding which to the first $k-1$ edges in L' doesn't create a cycle with them (**) and moreover results in reducing the number of forests.

This edge is the one we are looking for. When this edge e is added to T' it creates a cycle as T' is a tree. Also this cycle involves an edge other than the first $k-1$ edges in T' since e didn't create a cycle with the first $k-1$ edges of L' (**).

Hence, now using the arguments in the previous problem, we can remove the maximum weighted edge and what we've done in essence is that we've replaced the maximum weight edge in the cycle whose weight is $\geq L'[k]$ with e whose weight is $< L'[k]$. Therefore we obtain tree which is more optimal than the MST T'. A contradiction!

23.1-10

Notation. For tree $T = (V_T, E_T)$, $w(T) := \sum_{e \in E_T} w(e)$ that is it is the sum of the weights of the edges in T.

Proof Idea: Proof by contradiction. Note T is a spanning tree for G with weight function w' and $w'(T) = w(T) - k$.

Suppose T is not a MST for G with w' and hence there exists a tree T' which is an MST for G with w'. So $w'(T') < w'(T)$ $(*)$.

We have two cases to analyse. Firstly, if $(x, y) \in T'$ then $w'(T') = w(T') - k$ and from the above we have $w(T') < w(T)$. Now T' is a spanning tree for G with w which betters T which is a MST for G with w. A contradiction. Case 2, suppose $(x, y) \notin T'$ then $w'(T') = w(T')$ and so by $(*)$ we have $w(T') < w'(T) = w(T) - k$. A similar contradiction again!

Hence T is a MST for G with w'.

23.2-7

Idea: We essentially use the idea from Problem 23.1.5. We can remove the maximum weighted edge from a cycle to obtain a better tree.

Algorithm: Let e be the lightest edge across the cut $(V - v, v)$, where v is the new vertex added, then add e to the tree. Now, for each new edge (v, w) added in the graph do the following. Add the edge (v, w) to the tree, remove the maximum edge from the resulting cycle.
Finding the maximum weighted edge in a cycle: Follow the parent pointers from v to the root storing all the nodes encountered in list L_1. Do the same for w to get the list L_2. Of course, the last node on both L_1 and L_2 would

be the same, i.e. the root. Now, starting from the end of the lists, i.e. the root, keep moving backwards until you hit the first node, call it x, which is same on both the lists, this node is the least common ancestor of v and w. Finding the maximum edge between v and x, and w and x, and then comparing it with the new edge added would give us the maximum weighted edge in the cycle.

Correctness: Observe that the new tree obtained at each stage is the MST for the edges that we seen uptil now. The fact that the cycle resulting from the addition of an edge can be handled in the way done in the algorithm follows from an argument similar to that used in Problem 23.1.5.

Time Complexity: Finding the maximum weighted edge takes $O(|V|)$ time. Hence, the entire algorithm takes $O(k|V|)$ time, where k is the number of new edges added.

23-1

a.

Say $T = (V, E_T)$ is a spanning tree of $G = (V, E)$. Then $|E| \geq |E_T| = |V|-1$. Since it is given that $|E| \geq |V|$, we know that more than one spanning tree exists. And moreover MST and 2B-MST (Second-best minimum spanning tree) exists since the edge weights are distinct.

The uniqueness of the MST follows from Problem 23.1.8 and the fact that the weights of the edges are distinct.

We give an example where the second-best MST is not unique. Let $V = \{u, v, w, x\}$,

$$E = \{(u, v) : 1, (u, w) : 2, (v, x) : 5, (v, w) : 3, (w, x) : 4\}$$

the numbers adjoining the edges are their weights. The MST for the graph is $\{(u, v), (u, w), (w, x)\}$. However, there are two second-best MSTs:
$\{(u, v), (v, x), (v, w)\}$ and $\{(u, w), (v, w), (w, x)\}$.

b.

We want to show that a second-best minimum spanning tree can be obtained from the minimum spanning tree by replacing only one edge. Bear in mind that any spanning tree has the same number of edges.

Let $T_1 = (V, E_1)$ be a MST of G, and $T_2 = (V, E_2)$ be a 2B-MST of G and *we assume that T_2 has as many edges in common with T_1 as possible* $(***)$.

Notation. $E_1 - E_2$ is the set of edges which are in E_1 but not in E_2.

We assume that $T_1 \neq T_2$ and so $|E_1 - E_2| \neq 0$. We have two cases to analyze.

(i) If $|E_1 - E_2| = 1$, then we are done as we can transform T_1 to T_2 by replacing the edge which is in T_1 and not in T_2 by the edge which is in T_2 and not in T_1.

(ii) If $|E_1 - E_2| > 1$, then we show that we get a contradiction.

For $(u, v) \in E_1 - E_2$, define the cycle $C(u, v)$ by adding (u, v) to T_2. Let $e(u, v)$ be the edge of maximum weight in the cycle $C(u, v)$ and belongs to $E_2 - E_1$. Consider the cut $(S, V - S)$ that is formed by removing (u, v) from T_1, (let's say u and all nodes in the remaining subtree connected to u are in S, while v and all nodes in the other subtree connected to v are in $V - S$). We know that (u, v) is a light edge crossing this cut because it is in T_1. But (u, v) belongs to a cycle, namely $C(u, v)$. So, there is at least one more crossing edge beside (u, v) that belongs to $C(u, v)$.

Hence, (u, v) cannot be heavier than all edges in $C(u, v)$. Therefore, $w(u, v) < w(e(u, v))$ (since edge weights are distinct). Now, construct the tree $T_3 = T_2 - e(u, v) \cup (u, v)$.

By this construction, T_3 is a spanning tree of G and $w(T_3) < w(T_2)$.

And $T_1 \neq T_3$ so $w(T_1) < w(T_3) < w(T_2)$. So T_3 is a 2B-MST which has more edges in common with T_1 than T_2 has. This is a contradiction due to $(***)$. Hence $|E_1 - E_2| \not> 1$ and we are left with case (i).

c.

Starting from each node v do a DFS, labeling each node. Each node, x, adjacent to v has the label $w(v, x)$. Any other node y, has the label $max\{label(p(y)), w(p(y), y)\}$, where $p(y)$ is the parent of y in the DFS tree. The label on a node x gives $max[v, x]$.

DFS on a graph takes $O(|V|+|E|)$ time. Performing DFS on a tree takes $O(|V|)$ since $|E| = |V| - 1$.

Since, we are doing DFS on each node, the time complexity of the algorithm is $O(|V|^2)$.

d.

Using the $max[u, v]$ computed in the previous part, simply find an edge (u, v) not in the MST such that $w(u, v) - w(max[u, v])$ is the smallest.