CS 423: Theory of Algorithms
Spring, 1999
Tarjan

**Problem Set 5 Solutions**

---

## Solutions to Problem Set No. 5 *(Corrected)*

1. (Heaps with addition) The solution to this problem is based on the idea in Problem Set 3, No. 2. In addition to the standard $F$-heap data structure, we combine the trees representing an $F$-heap into a single tree by linking their roots together. These extra links are called *top links*. For each heap, we keep track of all the top links in a circular list. Each node has a value, such that the sum of values along the path from the node to the root is its key. Whenever we do a link of two nodes $x$ and $y$, whether it is an ordinary link or a top link, if $y$ becomes a child of $x$, we update the value in $y$ by subtracting the value in $x$; this preserves keys. Inversely, whenever, we break a link between a node $x$ and a child $y$, we update the value in $y$ by adding the value in $x$, again preserving keys. Each heap is represented by a single tree, whose root has minimum key. To meld two heaps, we merely link their tree roots (via a top link), catenate their lists of top links, and add the new top link to the list. To perform *add-to-keys* $(k, h)$, we add $k$ to the value in the root of heap $h$. To insert a new item, we create a new, single-node tree, and link it to the root of the appropriate heap, via a top link that is added to the list of top links. To perform *delete-min* $(h)$, we break all top links of $h$ and all links of the original root of $h$ to its children. We return the item in the original root. We repeatedly link remaining roots of equal rank until no such linking is possible, and then link the roots of the remaining trees via top links while creating a new list of top links for the updated heap. Add-to-keys takes $O(1)$ time, and all the other operations have the same amortized time bounds as for ordinary $F$-heaps.

   *Remark:* This data structure does *not* support *decrease key* in $O(1)$ amortized time, nor arbitrary deletion in $O(\log n)$ amortized time, because evaluating the key of a deleted item or an item whose key is decreased can be expensive. Handling these operations seems to require a separate disjoint set data structure to keep track of keys (using exactly the idea in Problem Set 3, No. 2), at an extra cost of inverse Ackerman function time per heap operation.

2. (Directed minimum spanning trees) I will describe an $O(m \log n)$-time solution. First, we modify the algorithm as described in the problem. For each vertex $v \neq r$, we may or may not have computed a cheapest incoming edge. Initially no $v \neq r$ has its cheapest incoming edge computed. The first phase of the algorithm stops when every evertex $v \neq r$ has its cheapest incoming edge computed. The second phase is the same as Step 3 of the problem description; it successively expands shrunken cycles in reverse order of their contraction to build a spanning tree of the original graph. The general step of phase 1 is a follows:

   Select a vertex $v \neq r$ with no incoming cheapest edge. Select a cheapest edge $(u, v)$ into $v$.

If $(u, v)$ forms a cycle with other selected edges, shrink this cycle into a single new vertex $x$ as in Step 2 of the problem description. The new vertex $x$ does not yet have an incoming selected edge.

There are at most $2n - 3$ iterations of the general step. We can show this as follows. Let $\Phi$ be the number of selected edges plus $2n$ minus twice the current number of vertices. $\Phi$ is zero initially. If $k$ is the current number of vertices, $\Phi$ is at most $k-1+2n-2k = 2n-k-1 \leq 2n-3$, since $k \geq 2$ (no vertex is shrunk into $r$). Each iteration of the general step increases $\Phi$ by at least one: one iteration adds one selected edge and may shrink a cycle of $\ell \geq 2$ vertices, which eliminates $\ell$ selected edges but also eliminates $\ell - 1$ vertices, and increases $\Phi$ by $1 + 2(\ell - 1) - \ell = \ell - 1 \geq 1$.

To implement this method, we use a disjoint set data structure to keep track of the shrunken vertices. Each set is a set of original vertices that have been shrunk into a single vertex; the name of the set is the current vertex into which all of these original vertices have been shrunk. For convenience, let us name the original vertices $1, 2, \ldots, n$, and name the new vertices formed by shrinking $n+1, n+2, \ldots$. (Then the maximum vertex name is $2n - 2$.) Given any original edge $(v, w)$, we can compute the corresponding current edge as $(\mathit{find}(v), \mathit{find}(w))$. The total number of finds will be $O(m)$, and the total time for finds will be $O(m\alpha(n)) = (m \log n)$. (Actually, the total find time is $O(n \log n + m)$; see Problem Set 2, No. 3.)

For purposes of selecting minimum-cost edges, we construct, for each vertex, a heap of incoming edges, with costs as keys. We use the data structure of Problem 1. When contracting a cycle, we use *add-to-keys* to update the keys of each heap corresponding to a vertex on the cycle, and then we meld these heaps. There are a total of $O(n)$ *add-to-keys* and *meld* operations.

In order to perform minimum-cost edge selection efficiently, we need another data structure. The selected edges define a set of rooted trees. We maintain an additional disjoint set data structure, whose items are the original graph vertices, grouped into sets based on the trees containing the corresponding current vertices. The name of each set is the root of the corresponding tree (a current vertex). We call this structure the *superset structure* (in contrast to the *set structure*, used to group original vertices into those corresponding to current vertices); we denote by *sfind* the find operation on the supersets.

Our last data structure is just the set of all current vertices not having a selected edge. When this set becomes empty, we stop the first phase. To select a minimum-cost edge, we choose a current vertex $v$ from the set. We do successive *delete-min* operations from the heap associated with $v$. Such a delete-min gives an original edge $(x, y)$ such that $\mathit{find}\ (y) = v$. If *find* $(x) = v$, we discard the edge and do another *delete-min*; $(x, y)$ corresponds to a loop. Once we have an edge $(x, y)$ with *find*$(x) \neq v$, we have found a valid edge for selection. If *sfind* $(x) \neq$ *sfind*$(y)$, the edge does not form a cycle. In this case we select the edge and unite the supersets with names *sfind*$(x)$ and *sfind*$(y) = v$. If *sfind*$(x) =$ *sfind*$(y)$, the selected edge does form a cycle. We determine the cycle by following selected edges backwards from $v$, and shrink the cycle, updating the keys and sets appropriately. Then we add the newly created vertex, say $z$, to the set of vertices without a selected edge.

This is enough to perform phase 1 in $O(m \log n)$ time, since there are at most $m$ *delete min* operatons, one per edge. Note that this algorithm does not explicitly delete multiple edges, nor does it delete loops until they surface during *delete min* operations, contrast to the algorithm description to the problem definition.

To perform phase 2 efficiently, we need to build one more data structure during phase 1, which keeps track of the cycles shrunk. This structure consists of a forest of rooted ordered trees, whose roots are the current vertices. Initially each original vertex is in a single-node forest. When we shrink a cycle consisting of vertices $v_1, v_2, \ldots v_k$ to form a new vertex $z$, we add $z$ as a new vertex to the forest, and we make $v_1, v_2, \ldots, v_k$ its children, in the order they occur along the cycle. We also store with each $v_i$ the original edge corresponding to the selected edge entering $v_i$ that is on the cycle. Once this structure is built during phase 1, we use it to perform phase 2, as follows. By numbering the vertices in both preorder and in postorder, we can test the ancestor-descendent relationship in $O(1)$ time. This allows us to test, for any two vertices $x$ and $y$, whether $x$ is a vertex into which $y$ was shrunk (possibly via several cycle-shrinkings). After phase 1, each root of a tree in the forest except for $r$ has an incoming selected edge. We expand all the shrunken cycles by processing each tree from root to leaves. Given a vertex $z$ with children $v_1, v_2, \ldots, v_k$, with original edge $(x, y)$ corresponding to the selected edge into $z$, we determine which of $v_1, v_2, \ldots, v_k$ is an ancestor of $y$. Say it is $v_i$. We make $(x, y)$ the selected edge for $v_i$ and for each $v_j \neq v_i$ we make the original edge corresponding to the cycle edge into $v_j$ its selected edge. Once all the cycles are expanded, the selected edges give a minimum spanning tree in the original graph.

This algorithm can be simplified somewhat by using a depth-first strategy to select vertices for processing in phase 1. This eliminates the need for the superset data structure. The algorithm maintains only two trees that are not single nodes, one rooted at $r$, the other rooted at the vertex $v$ currently being processed. Current vertices are labeled as being in one of three states: in the tree rooted at $r$, in the current tree, or in neither tree. We also maintain a set of the current vertices in the current tree. The root of the current tree is cosen for processing. If the selected edge connects this tree to the tree rooted at $r$, the entire current tree is labeled as being part of the tree rooted at $r$, and a new vertex, if any exists, is chosen to be the root of a new current tree. If the selected edge connects two vertices in the current tree, it forms a cycle, which is shrunk into the new root of the current tree. Otherwise, it connects to a new vertex, which becomes the new root of the current tree.

Finally, I note that the running time can actually be reduced to $O(n \log n + m)$ by using the depth-first strategy described above and using additional structures and additional heap operations to handle loops and multiple edges efficiently. See H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected graphs," *Combinatoria* **6** (1986), 109–122.

3. (Heuristic Search)

(a) Let $v = v_1, v_2, \ldots, v_{k+1} = t$ be any path from $v$ to $t$. Summing the inequalities $e(v_i) - e(v_{i+1}) \leq c(v_i, v_{i+1})$ for $i = 1, 2, \ldots k$ and using $e(t) = 0$, we have $e(v) \leq \sum_{i=1}^{k} c(v_i, v_{i+1})$. Thus $e(v)$ is a lower bound on the length of any path from $v$ to $t$, and hence on the length of a shortest path. Summing the same inequalities around a cycle $v_1, v_2, v_3, \ldots, v_{k+1} = v_1$ gives $0 \leq \sum_{i=1}^{k} c(v_i, v_{i+1})$, which means that if $e$ is a safe estimate, every cycle has non-negative length. Thus if there is any negative cycle, and in particular any negative cycle from which $t$ is reachable, then no estimate can be safe.

(b) I claim that the algorithm maintains the invariant that, if $x$ is any vertex removed from $L$ and $y$ is any vertex not yet removed from $L$, then $d(x) + e(x) \leq d(y) + e(y)$. Furthermore, no vertex removed from $L$ can have its $d$-value updated, and hence it cannot be reinserted into $L$. These invariants hold initially. Suppose the invariants hold just before a vertex $y$ is deleted from $L$. When $y$ is deleted, $d(x) + e(x) \leq d(y) + e(y)$ for all vertices previously deleted from $L$. For any edge $(y, z), c(y, z) \geq e(y) - e(z)$ since $e$ is a safe estimate. Thus $d(y) + c(y, z) \geq d(y) + e(y) - e(z) \geq d(z)$ if $z$ was previously deleted from $L$, which means that scanning $y$ cannot reinsert any vertices into $L$. Also, if $z$ is a vertex whose $d$-value is updated as a result of scanning $y$, we have $d(z) + e(z) = d(y) + c(y, z) + e(z) \geq d(y) + e(y)$, which means that the other invariant is preserved.

(c) For the desired result to be true, we need to break ties correctly. Specifically, we assume that if $t$ is a candidate to be deleted from $L$, it will be deleted before any other vertex $v$ with an equal value of distance plus estimate. Then, for a safe estimate $e$, the analysis in part (b) shows that the vertices that are removed from $L$ are exactly $s, t$, and every vertex $v$ such that

$d(v)+e(v) < d(t)$, where $d(v)$ is the actual shortest distance from $s$ to $v$. If $f$ is a safe estimate such that $e(v) \geq f(v)$ for every vertex $v$, then every vertex satisfying $d(v) + e(v) < d(t)$ also satisfies $d(v) + f(v) < d(t)$, which means that if $v$ is deleted from $L$ when using estimate $e$, then $v$ will be deleted from $L$ when using estimate $f$.

(d) Since every edge has non-negative cost, the estimate $f(x, y) = 0$ is a safe estimate. Since every edge has cost at least one and all edges connect vertices whose coordinates differ by at most one, the estimate $f(x, y) = \max\{|x|, |y|\}$ is also safe. (For any pair of adjacent vertices, the $f$-estimates differ by at most one.)

Assume $(a, b) \neq (0, 0)$. Since the actual distance from $(a, b)$ to $(0, 0)$ is at most $|a| + |b|$, the only vertices removed from $L$ for any safe non-negative estimate, including $e$ and $f$, are those at distance at most $|a| + |b|$ from $(0, 0)$, of which there are $O((|a| + |b|)^2) = O(\max\{|a|, |b|\}^2)$. Since each vertex is of constant degree, this is also an upper bound on the number of edges explored by the algorithm. The running time is dominated by the time required for deleting minimum-key vertices from $L$ (assuming use of an $F$-heap). The total time is $O(\max\{|a|, |b|\}^2 \log \max\{|a|, |b|\}^2)$. For the estimate $f$, and hence by ($c$) also for the estimate $e$, this bound is tight to within a constant factor, since if $a = 0$ and all diagonal edges have cost exactly 1, heuristic search will visit at least $\Omega(|b|^2)$ vertices; namely, $(i, |b| - j)$ for $0 \leq j \leq |b|/2, |i| \leq j$.

The problem becomes more interesting if we assume that each diagonal edge has cost at least $1 + \epsilon$ for some fixed $\epsilon > 0$. Then a heuristic search with the zero estimate can still visit $O(\max\{|a|, |b|\}^2)$ vertices, but a heuristic search with estimate $f$ can visit only $O(|a| \cdot |b|)$ vertices, which will be much smaller if the ratio $\max\{|a|, |b|\}/ \min\{|a|, |b|\}$ is large. Exercise: prove this.