Homework 6

Fundamental Algorithms, Fall 2001, Professor Yap

- DUE: Mon Dec 10, in class.
- ANNOUNCEMENT: This is the last graded homework. We will provide one more UNGRADED homework before the Final Exam. But solutions will be provided, so that you can check your own answers and discuss with us.
- Final Exam is likely to be Monday Dec 17, in class, but this awaits confirmation from department.

---

1. [**15 POINTS**] Problem 23.2-1, page 573. Kruskal's algorithm for MST.

   ANSWER:

   Suppose we have a minimum spanning tree T of G. We want to show that our algorithm finds it if we are careful when we sort the edges. The only freedom we have in sorting is the ordering among edges with the same weight. To unsure that every edge in T is selected by Kruskal's algorithm, we simply place any edge of T ahead of the edges that are NOT in T but with the same weight.

2. [**15 POINTS**] Do a hand simulation of Dijkstra's algorithm on the undirected graph in figure 1. Please use vertex A is the source. Recall that this amounts to updating a single array $d[1..n]$ (or, $d[A, B, \ldots, F]$) indexed by the vertices.

   HINT: Try to follow conventions described in class. Display a matrix where the ith row is the value of $d[1..n]$ in stage i. When a node first enters the set S, write an underscore (or, circle) that value. You need not copy values to the next row if they are unchanged.

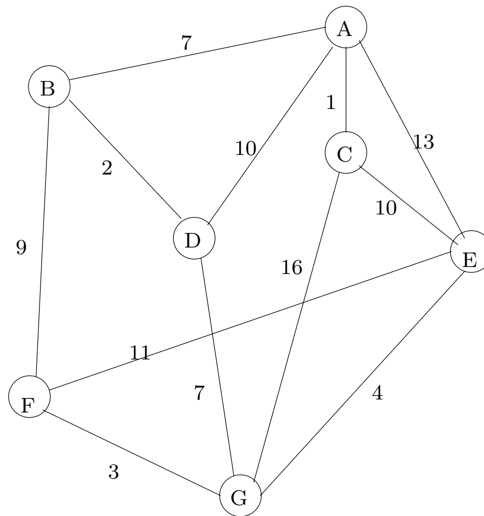


Figure 1: Graph with edge costs.

ANSWER:

	A	B	C	D	E	F	G
1	$\underline{0}$	7	1	10	13	∞	∞
2			$\underline{1}$		11		17
3		$\underline{7}$		9		16	
4				$\underline{9}$			16
5					$\underline{11}$		15
6							$\underline{15}$
7						$\underline{16}$	

3. [**15 POINTS**] Problem 22.1-6, p.530. An $O(n)$ algorithm to test for universal sink in a digraph. HINT: use induction on the number of vertices. At each maintain at most one candidate for a universal sink.

   ANSWER:

   Let $A(i,j)$ be the adjacency matrix of graph G on n vertices. If G' is the graph on the first $n-1$ vertices, let v' be the only candidate sink in G'. Let n be the nth vertex. By checking $A(v',n)$ and $A(n,v')$ we can eliminate at least one of v' and n. Thus the induction continues. What if G' has no candidates? Then, without even looking at A, we simply let n be the candidate.

   When we finally have a candidate v for G, we check in $2n$ more probes to confirm or reject that v.

4. [**15 POINTS**] Assume the path-compression heuristic but not the rank heuristic in this problem. Construct a sequence of 12 Link/Find operations on the nodes a,b,c,d,e such that the total cost is as large as possible. The cost of a Link is 1 and the cost of a Find is the number of nodes along a Find-path. You need not prove that your sequence is the best possible – so do this on a trial-and-error basis.

   NOTE: when performing $Link(x,y)$, you are allowed to make x or y the new root (in the absence of the rank heuristic).

   ANSWER:

   Initially we have:

   $$\{a\},\ \{b\},\ \{c\},\ \{d\},\ \{e\}.$$

   Here are 12 steps to achieve a cost of 26.

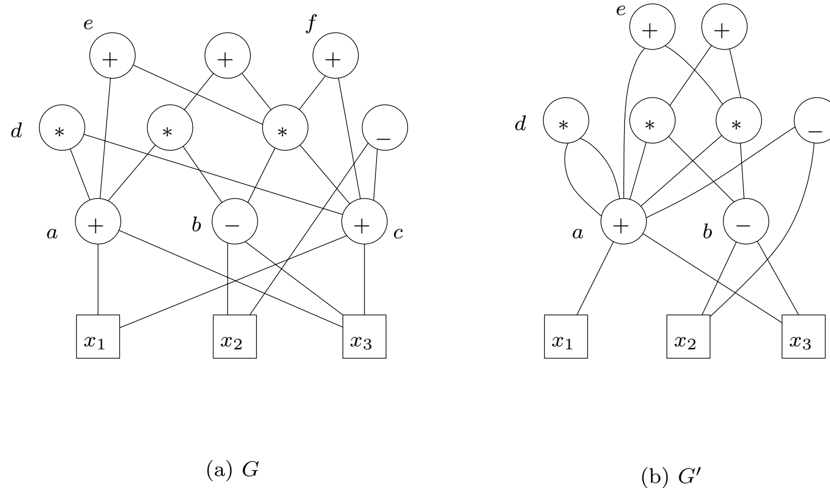
	Operation	Result	Cost
1	Link(a,b)	a | b	1
2	Link(a,c)	c | a | b	1
3	Find(b)	c / \\ a b	3

	Operation	Result	Cost
4	Link(d,c)	d(c(a, b))	1
5	Find(b)	d(c(a), b)	3
6	Find(a)	d(a, b, c)	3
7	Link(e,d)	e(d(a, b, c))	1
8	Find(a)	e(a, d(b, c))	3
9	Find(b)	e(a, b, d(c))	3
10	Find(c)	e(c, a, d, b)	3
11	Find(a)	e(c, a, d, b)	2
12	Find(a)	e(c, a, d, b)	2

Note that we do not prove that 26 is the worst case cost, but it probably is. Trying to prove this is basically a tedious case analysis.

5. [**15+15+20+15 POINTS**] Suppose you are given a directed acyclic graph ("DAG") G represented as an adjacency list. The nodes with indegree 0 are called sources, and the nodes with outdegree 0 are

called sinks. Each sink of G is labeled with a distinct variable name (x_1, x_2, etc). Each non-sink node of G has outdegree 2 and are labelled with one of four arithmetic operations $(+, -, *, \div)$. See figure 2, where the directions of edges are implicitly from top to bottom. Thus, every node of G represents an algebraic expression over the variables. E.g., node b represents the expression $x_2 - x_3$.



(a) G

(b) G'

Figure 2: Two DAGs: G and G'

Furthermore, the two edges exiting from the $-$ and $\div$ nodes are distinguished (i.e., labeled as "Left" or "Right" edges) while the two edges exiting from $+$ or $*$ nodes are indistinguishable. This is because $x - y$ is different from $y - x$, and $x \div y$ is different from $y \div x$, while $x + y = y + x$ and $x * y = y * x$. Also, this is actually a multigraph because it is possible that there there are two edges from a node d to another node a (e.g., see the graph G' in figure 2(b). Also, the Left/Right order of edges are implicit in our figures.

We define two nodes to be **equivalent** if they are both internal nodes with the same operator label, and their "corresponding children" are identical or (recursively) equivalent. This is a recursive definition. By "corresponding children" we mean that the order (left or right) of the children should be taken into account in case of $-$ and $\div$, but ignored for $+$ and $*$ nodes. For instance, the nodes a and c in figure 2(a) are equivalent. Recursively, this makes e and f equivalent.

Our goal is to construct a **reduced DAG** G' from G in which each equivalence class of nodes in G are replaced by a single new node. For instance with G in figure 2(a) as input, the reduced graph is G' in figure 2(b).
(a) Describe an efficient algorithm to compute the height of each node of G. Height is defined as the length of longest path to a sink. E.g., height of node e in figure 2(a) is 3 (there are paths of lengths 2 and 3 from e to sinks).
(b) Briefly argue why your algorithm is correct and analyze its running time.
(c) Design an efficient algorithm to compute the reduced DAG for any input G. HINTS: To "merge" equivalent nodes, use Union-Find. Note that you only need to check for equivalence among nodes of the same height, and this must be done in a bottom-up manner (i.e., smaller height nodes are merged first). To avoid quadratic behavior, try sorting.
(d) Analyze the running time of your algorithm.

ANSWER:

(a) Use DFS. Assume G is represented by adjacency lists. Initially color all nodes u as "unseen" and assign $h(u) = 0$. For each node u in the graph, if u is "unseen", call $DFS(u)$. We define $DFS(u)$ as follows:

```
DFS(u)
    Color u "seen".
    For each v adjacent from u,
        If v is "unseen", DFS(v)
        h(u) = max{h(u), 1 + h(v)} (*)
```

(b) The running time is $O(m+n)$ as in standard DFS. Why is this correct? It is enough to show that $h(u)$ is the height at the end. It is clear that sinks have $h(u) = 0$ since there are no nodes that are adjacent FROM a sink. Inductively, if every node v of height less than k has the correct value of $h(v)$, then the assignment to $h(u)$ in (*) is correct.

(c) Using part(a), we can assume that all the nodes of the same height are put in a linked list. More precisely, if the maximum height is H, we construct an array $A[1..H]$ such that $A[i]$ is a linked list of all nodes with height i. We use a union/find data structure for the nodes, where initially all the sets are singleton sets. Whenever we discover two nodes equivalent, we form a union. We will now process the list $A[i]$ in the order $i = 1, 2, \ldots, H$. To process the list $A[i]$ we proceed as follows: for each u in $A[i]$, we first find their children u_L and u_R using the adjacency lists of graph G. Then we perform $U_L = find(u_L)$ and $U_R = find(u_R)$. Hence U_L and U_R are the representive nodes of their respective equivalence classes. At this point, we do a trick: let $op(u)$ be the operator at node u. In case $op(u)$ is $+$ or $*$, we compare U_L and U_R (all nodes can be regarded as integers). If $U_R < U_L$, we swap their values. We now construct a 3-tuple

$$(op(u), U_L, U_R)$$

which serves as the key of u. Finally, we sort the list $A[i]$ using the keys just constructed. Since the keys are triples, we compare keys in a lexicographic order. We can assume an arbitrary order for the operators (say $'+' <' -' <' *' <' \div'$). Two nodes are equivalent iff they have identical keys. After sorting, all the nodes that are equivalent will be adjacent in the sorted list $A[i]$. So, we just go through the sorted list, and for each u in the list, we check if the key of u is the same as that of the next node v in the list. If so, we perform a $merge(u, v)$.

(d) Complexity: DFS is $O(m+n)$. Union Find is $m\alpha(n)$. Sorting of all the lists is $O(n \log n)$. Hence the overall complexity is $O(m + n \log n)$.

The following exercises are NOT to be handed in, but we encourage you to try to solve them.

1. Repeat problem 4, this time assuming the rank heuristic, but not path compression.

   ANSWER:

   In this case, simply build up the binomial tree B_2 (with 3 links) and do 9 finds on the deepest node of B_2. Each find costs 3, so the total cost is 30. Note that one of the 5 nodes is simply ignored in this lower bound! Again, we do not give a proof that this is the worst case, but it may well be.

2. Problem 23.2-8, page 574. Divide and Conquer algorithm for MST?

   ANSWER:

   Professor Toole is wrong. There is a trivial counter example with 3 nodes. Try it!

3. Problem 24.3-2, page 600. Dijkstra's algorithm for negative weights?

   ANSWER:

   It is simple to give a counter example with just 3 nodes. Try it!