15–210: Parallel and Sequential Data Structures and Algorithms

Exam II (Solutions)

Novermber 8, 2012

- There are 10 pages in this examination, comprising 5 questions worth a total of 100 points. The last 2 pages are an appendix with costs of sequence, set and table operations.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question. Clearly indicate your answers.
- You may refer to your one double-sided $8\frac{1}{2} \times 11$in sheet of paper with notes, but to no other person or source, during the examination.
- Your answers for this exam must be written in blue or black ink.

Full Name: Edsger W. Dijkstra

Andrew ID: _____ Section: _____

Question	Points	Score
Binary Answers	24	
Short Answers	14	
Ancestors	22	
Tree Contraction	22	
Dynamic Dijkstra	18	
Total:	100	

Question 1: Binary Answers (24 points)

Clearly write **True** or **False** to the left of each question.

(a) (3 points) If you can reduce your problem to comparison-based sorting in $O(n)$ work, then no algorithm exists to solve your problem in less than $\Theta(n \log n)$ work.

Solution: False

(b) (3 points) If you can reduce comparison-based sorting to your problem in $O(n)$ work, then no algorithm exists to solve your problem in less than $\Theta(n \log n)$ work.

Solution: True

(c) (3 points) If a graph has negative edge weights, the unmodified Dijkstra's algorithm for shortest path may loop forever.

Solution: False

(d) (3 points) If you add a constant to the weight on every edge in a weighted graph, it does not change the shortest paths.

Solution: False

(e) (3 points) Kruskal's algorithm for minimum spanning tree works with negative edge weights.

Solution: True

(f) (3 points) For independent random variables X and Y, $\max(\mathbf{E}[X], \mathbf{E}[Y]) = \mathbf{E}[\max(X, Y)]$.

Solution: False

(g) (3 points) Given the `split` and `join` operations for simple binary search tree described in class last Thursday (no balance criteria), the result of `join(split(A,k))` and `A` will always be the same tree for any A and k.

Solution: False

(h) (3 points) Given the `split` and `join` operations for a **Treap** described in class this week, the result of `join(split(A,k))` and `A` will always be the same tree for any A and k.

Solution: True

Question 2: Short Answers (14 points)

Please answer the following questions each with a few sentences, or a short snippet of code (either pseudocode or SML code). It has to fit in the given space. You will be graded on clarity as well as correctness.

(a) (7 points) Joe Sort conjectures that for quicksort with random pivots all keys are expected to be involved in the same number of comparisons. Let C_i be a random variable representing how many comparisons in which the i^{th} key in the sorted sequence is involved. Write down an expression for $E[C_i]$ (you don't need to solve it), and argue whether Joe is right or wrong.

Solution: Let C_{ij} be an indicator random variable that is 1 if i is compared to j and 0 otherwise. The i^{th} and j^{th} keys (in sorted order) are compared if either key is picked as a pivot before any of the keys between them. That is, $\mathbf{Pr}[C_{ij}] = 2/(|i-j|+1)$. Thus,

$$\begin{aligned}
\mathbf{E}[C_i] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\
&= \sum_{k=2}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&= 2(H_i + H_{n-i+1} - 2) \\
&< 2(\ln i + \ln(n-i+1))
\end{aligned}$$

Joe is wrong since this equation indicates that the median key will likely be involved in almost twice as many comparisons (about $4(\ln n - 1)$) as the smallest or largest ones (about $2 \ln n$ comparisons).

(b) (7 points) Suppose that you are given an undirected graph $G = (V, E)$ with weight function w and its minimum spanning tree T. Suppose that we add an edge e to G giving a graph $G' = (V, E \cup \{e\})$ and a new minimum spanning tree T'. What is the maximum number of edges by which T and T' can differ? Assume all edge weights are unique. Justify your answer.

Solution: 1. If the new edge is in T' then adding it to T would create a cycle. The maximum edge on that cycle would not be in T'.

Question 3: Ancestors (22 points)

Given a graph $G = (V, E)$, the traversal order of a depth-first search starting from s defines a tree $T(G, s)$, known as the *depth-first search tree*, with the following properties:

(i) s is the root of the tree $T(G, s)$;

(ii) the tree $T(G, s)$ contains exactly the vertices reachable from s; and

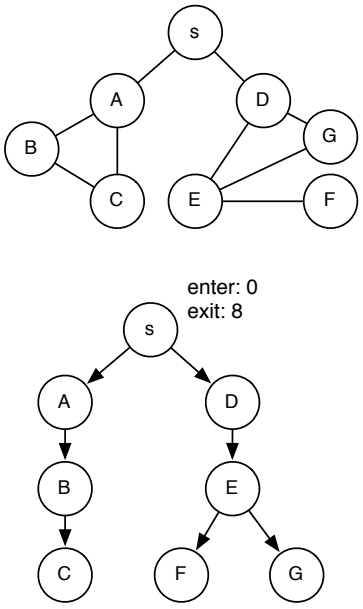
(iii) there is an *arc* (a directed edge) from u to v if the *first* visit to v is from u (i.e., u calls DFS on v and v hasn't been visited before).

In this problem, you'll design an algorithm that determines whether a vertex is an ancestor of another vertex in a depth-first search tree. A vertex u is an *ancestor* of a vertex v with respect to $T(G, s)$ if and only if there is a (directed) path from u to v. **Note:** Use the convention that u is an ancestor of itself.

(a) (5 points) Consider the following graph (left) and its DFS tree (right). Suppose we keep a counter (initially 0) and increment it every time an *enter* function is called. Specifically, we use a state of type `int`, initially with $S_0 = 0$, and the enter and exit functions

```
fun enter (s : int, v) = s+1
fun exit (s : int, v) = s
```

For each vertex in the tree below, fill in the corresponding row in the table with the values to which the *enter* and *exit* functions are applied for that vertex. For example, at the root node s, the counter is 0 when *enter* was called and is 8 at exit.



Solution:

Vertex	Enter Value	Exit Value
S	0	8
A	1	4
B	2	4
C	3	4
D	4	8
E	5	8
F	6	7
G	7	8

(b) (5 points) Let $\mathsf{enter}(x)$ denote the counter value c when *enter* is called on node x and $\mathsf{exit}(x)$ denote the counter value c when *exit* is called on x. If u is an ancestor of v, what is the relationship between their enter/exit values. Your answer should be of the form "u is an ancestor of v if and only if ....".

Solution: $\mathsf{enter}(v) \geq \mathsf{enter}(u)$ and $\mathsf{exit}(v) \leq \mathsf{exit}(u)$

In the following parts, you will write a preprocessing algorithm `make` that on input $G = (V, E)$ runs in $O((|V|+|E|)\log|V|)$ work and span, so that subsequent queries to `ancestor` can *each* be answered in $O(\log|V|)$ work and span. The input graph is represented as an adjacency set and has type `vertexSet vertexTable`.

The `make` algorithm will be based on DFS using appropriate initial state S_0 and *enter* and *exit* functions. For reference here is the code.

```
fun DFS (G : vertexSet vertexTable, s : vertex) : treeInfo =
let
  fun DFS' ((X : vertexSet , S : treeInfo), v : vertex) =
    if (v ∈ X) then (X, S)
    else let
      val S' = enter(S, v)
      val (X', S'') = iter DFS' (X' ∪ {v}, S') (N_G(v))
      val S''' = exit(S'', v)
    in (X', S''') end
in
  DFS'((∅, S_0), s)
end
```

Your answers in the following parts must be in SML, not pseudocode..

(c) (6 points) Using the insight in parts (a) and (b), you will implement `make` by defining the type `treeInfo`, the initial state `S0`, and the `enter` and `exit` functions. Note that you do not need to use the same state type as in part (a). You can use any of the table and set operations from the appendix, but `make` must run in $O((|V|+|E|)\log|V|)$ work and span,

```
type treeInfo = (int vertexTable * int vertexTable * int)

val S0 : treeInfo = (vertexTable.empty(), vertexTable.empty(), 0)

fun enter((enterT, exitT, cnt) : treeInfo, v : vertex) : treeInfo =

     (vertexTable.insert (fn (n,o) => n) (v,cnt) enterT, exitT, cnt+1)



fun exit((enterT, exitT, cnt) : treeInfo, v : vertex) : treeInfo =

     (enterT, vertexTable.insert (fn (n,o) => n) (v,cnt) exitT, cnt)


fun make(G : vertexSet vertexTable, s : vertex) : treeInfo =
let
  val(_,S) = DFS(G, s)
in
  S
end
```

(d) (6 points) Fill in the `isAncestor` algorithm (in SML) such that if `T` is the result of `(make G s)`, then `isAncestor T (u,v)` answers if u is an ancestor of v in the tree $T(G, s)$. Each `isAncestor` call must cost at most $O(\log|V|)$ work and span. You may assume that u and v are in $T(G, s)$.

```
fun isAncestor (tree : treeInfo) ((u,v) : vertex*vertex) : bool =
```

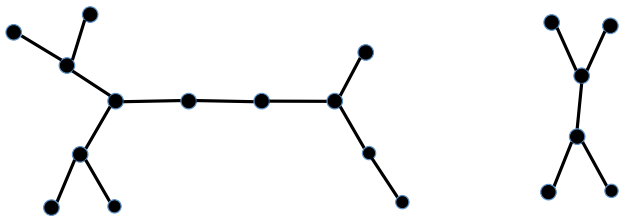
Solution:

```
let
  val (enterT, exitT, cnt) = tree
  val SOME(enterU) = find enterT u
  val SOME(exitU) = find exitT u
  val SOME(enterV) = find enterT v
  val SOME(exitV) = find exitT v
in
  (enterV >= enterU andalso exitV <= exitU)
end
```

Should check if u and v are found, but we accepted solutions without a check.

Question 4: Tree Contraction (22 points)

Consider an undirected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. We will call it a k-forest if it is a forest and all vertices have degree at most k. For example, the forest below is a 3-forest.



(a) (4 points) Recall that edge contraction takes an edge and contracts it so the two incident vertices (endpoints) are merged into a single vertex. To run edge contraction in parallel we want to pick a set of disjoint edges. Propose a technique for selecting disjoint edges in parallel in a k-forest, and analyze a lower bound on how many edges it will remove in expectation (trivial lower bounds will not get credit).

Solution: Pick a random number between 0 and 1 on each edge and choose the edge if it is the local maximum (larger than all edges sharing either endpoint). The edge will be removed with probability $2k - 1$ since that is the maximum number of edges it is competing with, including itself. Therefore the lower bound on the expected number that will be removed is $|E|/(2k - 1)$.

(b) (4 points) After one round of parallel edge contraction, will a k-forest necessarily still be a k-forest? Explain briefly.

Solution: No. By contracting an edge we could create a vertex of degree $2(k - 1)$.

(c) (4 points) Consider the following contraction rule:

1. For vertices of degree two, contract one of the two incident edges.
2. For vertices of degree one, contract the incident edge.

(Note that a disjoint set of such edges can be found by using the technique you proposed in part (a), except that an edge is considered for contraction only if at least one of its two incident vertices has degree one or two.) By only considering such contractions, will a k-forest still be a k-forest? Explain briefly.

Solution: Yes. Edge contraction with a degree two vertex will not change the degree, while edge contraction with a degree one vertex will reduce the degree of the combined vertex by 1.

(d) (6 points) For a 3-forest argue that if a constant fraction of the degree one and degree two vertices are removed then a constant fraction of the overall vertices are removed. Give a brief proof (a few sentences). Hint: Relate the number of degree 3 vertices with the number with degree 1 vertices.

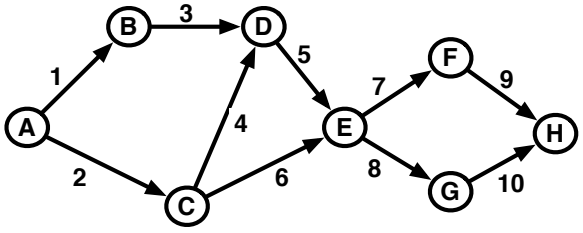
Solution: If there are l degree 3 vertices, there will be $l + 2$ degree 1 vertices. This can be proved by induction on the number of degree 3 vertices. It is true for a single degree 3 vertex since we have it in the middle and a bunch of degree 2 chains from each of its three neighbors that end in a degree 1 vertex. Therefore we have $l = 1$ and $l + 2 = 3$ degree 1 vertices. Now each time we add a degree three vertex we split a chain and add one more chain ending in a degree 1 vertex.

(e) (4 points) Now consider an algorithm for 3-forests that repeats the contraction on degree 1 and 2 vertices using the contraction rule in part (c) until no edges are left. What is the work and span of this algorithm? Explain in a few words.

Solution: For a forest with n vertices, the work is $O(n)$ since the size decreases geometrically and the span is $O(\log^2 n)$ since there are $O(\log n)$ rounds and each takes $O(\log n)$ span for the filter.

Question 5: Dynamic Dijkstra (18 points)

Consider the graph shown below. Each edge is assigned a weight, which appears next to it. Suppose that you are travelling from A to H and you want to find the shortest path to your destination.



(a) (6 points) Show the order that the vertices are visited by Dijkstra and the distance computer from the source A.

A	1	0
B	2	1
C	3	2
D	4	4
E	5	8
F	6	15
G	7	16
H	8	24

short

(b) (3 points) What is the shortest path to H? Write it as a sequence of vertices.

Solution: $A\ C\ E\ F\ H$.

(c) (9 points) Suppose you have calculated the shortest path from A to H, and in the process have stored the distances to every other vertex. Now let's say someone adds a new road between D and F. You could recalculate the shortest paths by starting again at A. But you realize that this might be wasteful since not much changes.

More generally, consider a graph $G = (V, E, w)$, where w is a weight function from edges to non-negative reals. Assume you have already calculated the shortest path from some source s to all $v \in V$. Now assume that some arc e is added to the graph. This arc might change the distances from s to some set of vertices $U \subset V$ (in particular it might make the distances closer). Describe how to update all the shortest paths to $u \in U$ using some variant of Dijkstra's algorithm. Your variant should only require $O(\delta \log |V|)$ work, where $\delta = \sum_{u \in U} \text{degree}(u)$.

Solution: Suppose the edge added is $(s, y$ with weight $w(x, y)$. Initialize dijkstra to use the existing table of distances, and insert $(d[x]+w(x, y), y)$ onto the priority queue. Modify dijkstra to revisit vertices. That is, when it dequeues, if the distance on the queue is greater than or equal to the distance in the table, it simply recurses without visiting the vertex. Otherwise, it updates the distance in the distance table and relaxes all its neighbors. That is, it computes a new distance for each neighbor and puts it on the priority queue. Since it only revisits vertices that have revised distances, it only adds neighbors of U to the priority queue. Each of these δ vertices requires a lookup and a possible update in the distance table, which costs $O(\log |V|)$ work.