Full Name: _____

Andrew ID: _____ **Section:** _____

15–210: Parallel and Sequential Data Structures and Algorithms

Practice Exam II (Solutions)

April 2014

- There are 16 pages in this examination, comprising 7 questions worth a total of 120 points. The last few 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.

Circle the section YOU ATTEND

Sections		
A	9:30am - 10:20am	Naman
B	10:30am - 11:20am	Sam
C	12:30pm - 1:20pm	Isaac
D	12:30pm - 1:20pm	Nikki
E	1:30pm - 2:20pm	Esther and Ronald
F	1:30pm - 2:20pm	Ivan
G	3:30pm - 4:20pm	Will and Ian

Full Name: _____ **Andrew ID:** _____

Question	Points	Score
Minimum Spanning Trees	10	
Graphs	15	
Short Answers	20	
Set Operations	15	
Strongly Connected Components	20	
MST and Tree Contraction	25	
Treaps	15	
Total:	120	

Question 1: Minimum Spanning Trees (10 points)

Suppose that you are given a graph $G = (V, E)$ and a its minimum spanning tree T. Suppose that we delete from G, one of the edges $(u, v) \in T$ and let G' denote this new graph.

(a) (3 points) Is G' guaranteed to have a minimum spanning tree?

Solution: No, the graph can become disconnected.

(b) (3 points) Assuming that G' has a minimum spanning tree T'. TRUE or FALSE: the number of edges in T' is no greater than the number of edges in T? Explain your answer in one sentence.

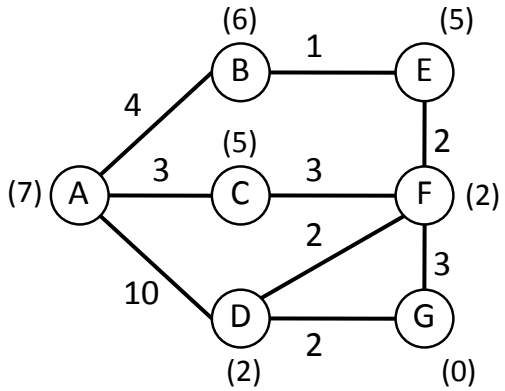
Solution: TRUE. A spanning tree always has $|V| - 1$ edges.

(c) (4 points) Assuming that G' has a minimum spanning tree T', describe an algorithm for finding T'. What is the work of your algorithm?

Solution: At each iteration, find the minimum weight edge that connects the two disconnected components. $O(m \log n)$.

Question 2: Graphs (15 points)

(a) (6 points) Consider the graph shown below, where the edge weights appear next to the edges and the heuristic distances to vertex G are in parenthesis next to the vertices.



i. Show the order in which vertices are visited by Dijkstra when the source vertex is A.

Solution: A C B E F D G

ii. Show an order in which vertices are visited by A^* when the source vertex is A and the destination vertex is G.

Solution: A C F G

(b) (4 points) What is the key reason you would choose to use A^* instead of Dijkstra's algorithm?

Solution: You can use A^* if you want the shortest path to only a single goal vertex, and not all shortest paths. A^* can be much more efficient, as it tries to move toward the goal more directly, skipping many more vertices.

(c) (5 points) Show a 3-vertex example of a graph on which Dijkstra's algorithm always fails. Please clearly identify which vertex is the source.

Solution:

```
        A
       / \
  x=4 /   \ y=-2   x+y < z < x guarantees failure
     /     \       x+y < z <= x may fail depending on the input order
    S ------- B
        z=3
```

Question 3: Short Answers (20 points)

Please answer the following questions each with a few sentences, or a short snippet of code (either pseudocode or SML).

(a) (5 points) Consider an undirected graph G with unique positive weights. Suppose it has a minimum spanning tree T. If we square all the edge weights and compute the MST again, do we still get the same tree structure? Explain briefly.

> **Solution:** Yes we get the same tree. The minimum spanning tree only depends on the ordering among the edges. This is because the only thing we do with edges is compare them.

(b) (5 points) A new startup *FastRoute* wants to route information along a path in a communication network, represented as a graph. Each vertex represents a router and each edge a wire between routers. The wires are weighted by the maximum bandwidth they can support. *FastRoute* comes to you and asks you to develop an algorithm to find the path with maximum bandwidth from any source s to any destination t. As you would expect, the *bandwidth* of a path is the minimum of the bandwidths of the edges on that path; the minimum edge is the *bottleneck*.

Explain how to modify Dijkstra's algorithm to do this. In particular, how would you change the priority queue and the following relax step?

```
fun relax (Q, (u,v,w)) = PQ.insert (d(u) + w, v) Q
```

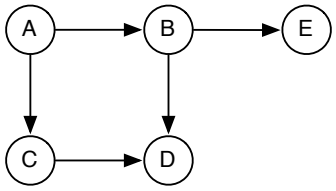
Justify your answer.

> **Solution:** We'll use a max priority queue instead of a min priority queue used in Dijkstra's. We will also modify the relax step to insert into the priority queue $\min(d(u), w)$ because the quality of a path is the minimum of the edge weights. These changes don't affect the correctness of Dijkstra's, so we could explore the vertices like in Dijkstra's.

(c) (5 points) Given a graph with integer edge weights between 1 and 5 (inclusive), you want to find the shortest *weighted* path between a pair of vertices. How would you reduce this problem to the shortest *unweighted* path problem, which can be solved using BFS?

Solution: Replace each edge with weight i with a simple path of i edges each with weight 1. Then solve with BFS.

(d) (5 points) Recall the implementation of DFS shown in class using the `enter` and `exit` functions. Circle the correct answer for each of the following statements, assuming DFS starts at A:



`enter D` could be called before `enter E`:	True	False
`enter E` could be called before `enter D`:	True	False
`enter D` could be called before `enter C`:	True	False
`exit A` could be called before `exit B`:	True	False
`exit D` could be called before `enter B`:	True	False

Solution: True, True, True, False, True

Question 4: Set Operations (15 points)

We can represent ordered sets of integers using binary search trees using the type:

```
datatype bst = Leaf | Node of (bst * bst * int)
```

This datatype is paired with the functions:

```
val split : (bst * int) -> bst * bool * bst
val join : (bst * int option * bst) -> bst
```

where `split(T,k)` returns a pair of trees from `T` (less than and greater than `k`), and a flag indicating if `k` is in `T`.

Joe Twoten noticed that the course staff kept writing almost the same code to implement set union, set intersection and set difference. He decided a more elegant solution would be to use higher order functions and just write the bulk of the code once. Typical of Joe, he only typed in part of the solution and left the rest up to you.

(a) (6 points) Consider a function

```
val combine : (bool * (int * bool -> int option)) -> (bst * bst) -> bst
```

where

- `combine (true,f) (Leaf,T2)` evaluates to `T2`
- `combine (false,f) (Leaf,T2)` evaluates to `Leaf`
- `combine (b,f) (T1,T2)` evaluates to a `bst` where every key `k` that appears in `T1` is replaced by the result of applying `f` to `k` and a boolean indicating whether `k` appears in `T2`. Every key that appears only in `T2` is handled as specified by `b` in the base case.

Most of the implementation of `combine` is provided below. Finish it by filling in the blanks.

```
fun combine (keep :  bool, f :  int * bool -> int option)
            (T1:  bst, T2:  bst) :  bst =
  case (T1, keep) of
      (Leaf, true) => T2

    | (Leaf, false) => ____________________
    | (Node(L1,R1,k1), _) =>
      let

        val (L2, exists, R2) = split (  T2, k1  )
      in

        join (  combine (keep, f) (L1, L2)  ,

              f(k1,exists)  ,

              combine (keep, f) (R1, R2)  )
      end
```

(b) (6 points) We can use `combine` to implement the various set operations by making one call with carefully chosen arguments. For example,

```
val union = combine (true,(fn (k,_) => SOME k)
```

You may assume that `combine` works correctly, even if you did not implement it.

i. Implement set intersection with exactly one call to `combine`.

```
val inter = combine (false,(fn (k, true) => SOME k
                  | (k, false) => NONE)
```

ii. Implement set difference with exactly one call to `combine`.

```
val diff = combine (false,(fn (k, true) => NONE
                  | (k, false) => SOME(k))
```

iii. Implement symmetric set difference with exactly one call to `combine`. Recall that the symmetric difference of sets A and B is

$$A \Delta B := \{x : x \in A \oplus x \in B\}$$

where $\oplus$ is exclusive or.

```
val symdiff = combine (true,(fn (k, true) => NONE
                  | (k, false) => SOME(k))
```

(c) (3 points) We could also implement `symdiff` as

```
fun symdiff (A,B) = diff(union(A,B), inter(A,B))
```

Give one reason (other than code reuse) that would make the implementation written with `combine` preferable.

Solution: Using combine would require passing over the trees just once instead of three times. It would therefore presumably be cheaper by an constant amount.

Question 5: Strongly Connected Components (20 points)

In this question, you will write 2 functions on directed graphs. We assume that graphs are represented as:

```
type graph = vertexSet vertexTable
```

with key comparisons taking $O(1)$ work.

(a) (10 points) Given a directed graph $G = (V, E)$, its transpose G^T is another directed graph on the same vertices, with every edge flipped. More formally, $G^T = (V, E')$, where

$$E' = \{(b, a) \mid (a, b) \in E\}.$$

Here is a skeleton of an SML definition for `transpose` that computes the transpose of a graph. Fill in the blanks to complete the implementation. Your implementation must have $O(|E| \log |V|)$ work and $O(\log^2 |V|)$ span.

```
fun transpose (G :  graph) :  graph =
  let
    val S = vertexTable.toSeq(G)       (* returns (vertex*vertexSet) seq *)

    fun flip(u,nbrs) = Seq.map  (fn v => (v,u))  (vertexSet.toSeq nbrs)

    val ET = Seq.flatten(Seq.map flip S)

    val T = vertexTable.  collect  ET
  in

    vertexTable.map   vertexSet.fromSeq   T
  end
```

(b) (10 points) A *strongly connected component* of a directed graph $G = (V, E)$ is a subset S of V such that every vertex $u \in S$ can reach every other vertex $v \in S$ (i.e., there is a directed path from u to v), and such that no other vertex in V can be added to S without violating this condition. Every vertex belongs to exactly one strongly connected component in a graph.

Implement the function:

```
val scc : graph * vertex -> vertexSet
```

such that `scc(G,v)` returns the strongly connected component containing v. You may assume the existence of a function:

```
val reachable : graph * vertex -> vertexSet
```

such that `reachable(G,v)` returns all the vertices reachable from v in G. Not including the cost of `reachable`, your algorithm must have $O(|E| \log |V|)$ work and $O(\log^2 |V|)$ span. You might find `transpose` useful and can assume the given time bounds.

```
fun scc (G :  graph, v :  vertex) :  vertexSet =

    vertexSet.intersection(reachable(G,v),

                           reachable(transpose(G,v)))
```

Question 6: MST and Tree Contraction (25 points)

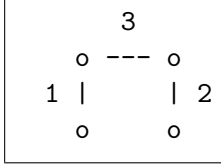
In *SegmentLab*, you implemented Borůvka's algorithm that interleaved star contractions and finding minimum weight edges. In this questions you will analyze Borůvka's algorithm more carefully.

We'll assume throughout this problem that the edges are undirected, and each edge is labeled with a unique identifier (ℓ). The weights of the edges do not need to be unique, and $m = |E|$ and $n = |V|$.

```
1  % returns the set of edges in the minimum spanning tree of G
2  function MST(G = (V, E)) =
3    if |E| = 0 then {}
4    else let
5      val F = {min weight edge incident on v : v ∈ V}
6      val (V', P) =contract each tree in the forest (V, F) to a single vertex
7                   V' = remaining vertices
8                   P = mapping from each v ∈ V to its representative in V'
9      val E' = {(P_u, P_v, ℓ) : (u, v, ℓ) ∈ E | P_u ≠ P_v}
10   in
11     MST(G' = (V', E')) ∪ {ℓ : (u, v, ℓ) ∈ F}
12   end
```

(a) (4 points) Show an example graph with 4 vertices in which F will not include all the edges of the MST.

Solution:



(b) (4 points) Prove that the set of edges F must be a forest (i.e. F has no cycle).

Solution: Answer 1: The MST does not have a cycle (it is a tree) and F is a subset of F so it can't have a cycle.

Answer 2: AFSOC that there is a cycle. Consider the maximum weight edge on the cycle. Neither of its endpoints will choose it since they both have lighter edges. Contradiction.

(c) (4 points) Suggest a technique to efficiently contract the forest in parallel. What is a tight asymptotic bound for the work and span of your contract, in terms of n? Explain briefly. Are these bounds worst case or expected case?

Solution: Use star contraction as described in class. Since in contraction a tree will always stay a tree, the number of edges must go down with the number of vertices. Therefore total work will be $O(n)$ and span will be $O(\log^2 n)$ in expectation.

(d) (4 points) Argue that each recursive call to `MST` removes, in the worst case, at least *half* of the vertices; that is, $|V'| \leq \frac{|V|}{2}$.

Solution: Every vertex will join at least one other vertex. Since edges have two directions, at least $n/2$ of them must be selected, which will remove at least $n/2$ vertices ($n = |V|$).

(e) (4 points) What is the maximum number of edges that could remain after one step (i.e. what is $|E'|$)? Explain briefly.

Solution: $m - n/2$ since at least $n/2$ edges are removed, as described in previous answer.

(f) (5 points) What is the expected work and span of the overall algorithm in terms of m and n? Explain briefly. You can assume that calculating F takes $O(m)$ work and $O(\log n)$ span.

Solution: Since vertices go down by at least a factor of $1/2$ on each round, there will be at most $\log n$ rounds. The cost of each round is dominated by calculating F, $O(m)$ work and $O(\log n)$ span and the contraction of forests $O(n)$ work and $O(\log^2 n)$ span. Multiplying the max of each of these by $\log n$ gives $O(m \log n)$ work and $O(\log^3 n)$ span.

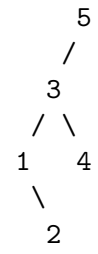
Question 7: Treaps (15 points)

(a) (5 points) Assume that priorities are generated using a random hash function $h : \mathsf{keys} \to [0, 1]$. For keys $1, 2, 3, 4, 5$, assume the corresponding hash values are as follows.

key	1	2	3	4	5
h(key)	0.4	0.1	0.5	0.2	0.6

What would the Treap look like if we insert the keys $1, 4, 2, 5, 3$ in this order?

Solution: Assuming a max heap (maximum priority at root):



(b) (5 points) In our analysis of the expected depth of a key in a Treap, we made use of the following indicator random variable:

$$A_i^j = \begin{cases} 1, & j \text{ is an ancestor of } i \\ 0, & \text{otherwise} \end{cases}$$

Write an expression for S_i, the size of a subtree rooted at key i, in terms of A_i^j.

Solution: $S_i = \sum_{j=1}^{n} A_j^i$

(c) (5 points) Derive a closed-form expression for $\mathbf{E}[S_i]$. You may use $\ln n$, H_n, or $n!$ in your expression.

Solution:

$$\begin{aligned} \mathbf{E}[S_i] &= \sum_{j=1}^{n} 1/(|j-i|+1) \\ &= H_i + H_{n-i+1} - 1 \\ &= O(\log n) \end{aligned}$$

Appendix: Library Functions

```
signature SEQUENCE =
sig
  type 'a seq
  type 'a ord = 'a * 'a -> order
  datatype 'a listview = NIL | CONS of 'a * 'a seq
  datatype 'a treeview = EMPTY | ELT of 'a | NODE of 'a seq * 'a seq

  exception Range
  exception Size

  val nth : 'a seq -> int -> 'a
  val length : 'a seq -> int
  val toList : 'a seq -> 'a list
  val toString : ('a -> string) -> 'a seq -> string
  val equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool

  val empty : unit -> 'a seq
  val singleton : 'a -> 'a seq
  val tabulate : (int -> 'a) -> int -> 'a seq
  val fromList : 'a list -> 'a seq

  val rev : 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
  val flatten : 'a seq seq -> 'a seq

  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val map2 : ('a * 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq
  val zip : 'a seq -> 'b seq -> ('a * 'b) seq

  val enum : 'a seq -> (int * 'a) seq
  val inject : (int * 'a) seq -> 'a seq -> 'a seq

  val subseq : 'a seq -> int * int -> 'a seq
  val take : 'a seq * int -> 'a seq
  val drop : 'a seq * int -> 'a seq
  val showl : 'a seq -> 'a listview
  val showt : 'a seq -> 'a treeview

  val iter : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b
  val iterh : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b seq * 'b
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val scan : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a seq * 'a
  val scani : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a seq

  val sort : 'a ord -> 'a seq -> 'a seq
  val merge : 'a ord -> 'a seq -> 'a seq -> 'a seq
  val collect : 'a ord -> ('a * 'b) seq -> ('a * 'b seq) seq
  val collate : 'a ord -> 'a seq ord

end
```

ArraySequence	Work	Span

`empty ()`
`singleton a`
`length s`
`nth s i`
	$O(1)$	$O(1)$

`tabulate f n`
if `f i` has W_i work and S_i span
`map f s`
if `f` s_i has W_i work and S_i span, and $\|s\| = n$
`map2 f s t`
if `f` (s_i, t_i) has W_i work and S_i span, and $\|s\| = n$
	$O\left(\sum_{i=0}^{n-1} W_i\right)$	$O\left(\max_{i=0}^{n-1} S_i\right)$

`reduce f b s`
if `f` does constant work and $\|s\| = n$
`scan f b s`
if `f` does constant work and $\|s\| = n$
`filter p s`
if `p` does constant work and $\|s\| = n$
`showt s`
if $\|s\| = n$
`hidet tv`
if the combined length of the sequences is n
	$O(n)$	$O(\lg n)$

`sort cmp s` *if* `cmp` does constant work and $\|s\| = n$	$O(n \lg n)$	$O(\lg^2 n)$

`merge cmp s t`
if `cmp` does constant work, $\|s\| = n$, and $\|t\| = m$
`flatten s`
if if $s = \langle s_1, s_2, \ldots, s_k \rangle$ and $m + n = \sum_i \|s_i\|$
	$O(m + n)$	$O(\lg(m + n))$

`append (s,t)` *if* $\|s\| = n$, and $\|t\| = m$	$O(m + n)$	$O(1)$

Table/Set Operations	*Work*	*Span*
`size`(T) `singleton`(k, v)	$O(1)$	$O(1)$
`filter` f T	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\lg\|T\| + \max_{(k,v)\in T} S(f(v))\Big)$
`map` f T	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\max_{(k,v)\in T} S(f(v))\Big)$
`tabulate` f S	$O\Big(\sum_{k\in S} W(f(k))\Big)$	$O\Big(\max_{k\in S} S(f(k))\Big)$
`find` T k `insert` f (k, v) T `delete` k T	$O(\lg\|T\|)$	$O(\lg\|T\|)$
`extract` (T_1, T_2) `merge` f T_1 T_2 `erase` (T_1, T_2)	$O\big(m \lg(\frac{n+m}{m})\big)$	$O\big(\lg(n+m)\big)$
`domain` T `range` T `toSeq` T	$O(\|T\|)$	$O(\lg\|T\|)$
`collect` S `fromSeq` S	$O(\|S\| \lg\|S\|)$	$O(\lg^2\|S\|)$
`intersection` (S_1, S_2) `union` (S_1, S_2) `difference` (S_1, S_2)	$O\big(m \lg(\frac{n+m}{m})\big)$	$O\big(\lg(n+m)\big)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$. For `reduce` you can assume the cost is the same as `Seq.reduce f init (range(T))`. In particular `Seq.reduce` defines a balanced tree over the sequence, and `Table.reduce` will also use a balanced tree. For `merge` and `insert` the bounds assume the merging function has constant work.