Fall 2013 Final Solutions (Beta Edition)

1. **TSTs.**
   a. `ID, IDD, IDDQD, IDKFA, XA, XX, XEA, XYZZY`
   b. `IDAAAA or XDAAAA or XCAAAA, or indeed anything that is six letters long and starts with ID, XD, or XC.`
2. **MSTs.**
   a. 1, 2, 3, 5, 6, 7, 9, 11, 12
   b. None. Adding the same value to all edges does not change their relative order, and that's all that matters for the MST.
   c. No. For example: A, B, C in a triangle, all with weight 5. Or almost any edge in the graph from part A that is not part of the MST.
3. **String Sorting.**
   a. 4 5 3 2 3 4 3
   b. Most obvious answer: Uses far too much memory (and space), requiring count arrays of size 2^{128}
   c. Most obvious answer: While the algorithm is linear, the W adds a large constant factor. How large? We will have WN charAt() operations vs. somewhere around 2.78 $N \lg N$ compareTo() operations (or less if our points tend to be distinct). This is not quite an apples to apples comparison, but 2.78 $\lg N$ is likely to be much smaller than 128 for real data sets.
4. **Regular Expressions**
   a. 0 → 1, 0 → 3, 2 → 10, 3 → 4, 3 → 8, 4 → 5, 5 → 4, 5 → 6, 7 → 9
   b. Weird NFAs
      i. A|B
      ii. (A*B)+
5. **Substring Search**
   a. `0 2 3 0 2 6 7 0 2`
      `1 1 1 4 5 1 1 8 9`
   b. `(Examined characters not shown)`

G	U	L	L	S	S	S	S	T	L	O	S	T	S	L	U	G	S	U	G
S	L	U	G	S															
		S	L	U	G	S													
			S	L	U	G	S												
				S	L	U	G	S											
									S	L	U	G	S						
													S	L	U	G	S		

   c. Tracking permutations requires you to either create a state for any possible permutation (factorial in size), or to create a bunch of parallel DFAs (factorial in number, but linear in size each).

   d. Solution coming soon.

6. **Directed Graphs**

a. MELTBANANA
b. FALSE. For example, $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E, Z \rightarrow Y \rightarrow X \rightarrow D, Z \rightarrow E$
c. Yes, no, no, no, no. It was also correct to say no for all problems since `edgeTo[v] = v` doesn't make sense.

7. Reductions

a. Yes, Linear.
b. Yes, Factorial. Since we failed to put factorial on the list of choices, exponential was acceptable.

8. Shortest Paths

a. B, 30.0, C 15.0, D, 13.0
b. F
c. F, 15.0
d. Yes, it works fine. In fact, relaxing additional edges can never cause any sequence of otherwise good relaxations to be suboptimal. Relaxation only decreases distances to the target vertex. In terms of our optimality conditions, relaxation of edge $e = v \rightarrow w$ can only decrease $distTo[w]$, so $distTo[w] \leq distTo[v] + e.weight()$ can never become incorrect due to any relaxation.

9. LZW compression

a. LOLO
b. Impossible, LZW would replace the second pairs of As with `0100`.
c. LOLOLOL
d. ZZZZZZ
e. Impossible, 102 cannot occur as the second codeword (as only one codeword has been generated at this point).

10. Max flow

a. Supply edges are 46, 45, 45, 34. Demand edges are 39, 50, 42, 38. Connecting edges in the middle can take any capacity larger than or equal to the corresponding supply vertex.
b. 168
c. The O vertices, the B vertices, and t.
d. No, yes, yes, no, no, no

11. Why did we do that again?

a. Natural representations from BSTs would all suffer from flaws. Representing in any kind of compact manner (i.e. ordered array or as the level order traversal) would require linear time to insert. Representing them in expanded heap-like array where the 0^{th} element represents the root, the 1^{st} element the left child fo the root (if it exists), and so forth would require tremendous amounts of space.
b. This question was not very clear. We took a wide variety of different answers, but the intention was that when a left parenthesis is removed from the stack, it is used to decide where to start one of the OR arrows, as well as where to end any following *'s backwards arrow.

c. BFS would still find shortest paths, since they would be dequeued by total distTo order, which is the exact same as the normal order. Performance would be either equal or worse, depending on the priority queue implementation. Using the array-heap based implementation from class, it would be worse, taking log time instead of constant time to handle each insert and dequeue operation.
d. The minimum increase in flow is 1. To prove convergence, we note that no flow can exceed any s-t cut. Since each s-t cut has a fixed capacity, eventually the flow must exceed that cut's capacity.
e. As long as edges are non-negative weight, cycles can never be shorter than the best path found so far.
f. Since the alphabet size is only 2, there is no need to use a TST. A TST would waste space and time (though it wouldn't be all that hard to implement, nor would its performance be much worse than a standard trie).

12. Design problems

a. One was allowed to assume that all paths have distinct lengths. The algorithm to follow works even if there are ties in path lengths, but then the problem becomes ambiguous: if there are two paths of the same shortest length, does one want to call one of them the shortest and the other the second-shortest, or does one want a path of the second-shortest length? If there are ties, the algorithm below will find the second-shortest length, but it can be modified to find two paths of the same length if there are such.

   The idea is to run an extension of Dijkstra's algorithm in which each vertex v has two tentative distances, $d1(v)$ and $d2(v)$, which are the two distinct shortest distances to v found so far. Critical to making this idea work is to allow a vertex v to be reinserted in the priority queue PQ once its actual shortest distance is known. No student solution mentioned this. In addition to distance, we need two parent vertices, $p1(v)$ and $p2(v)$, for each vertex v, which are the predecessor of v on its tentative shortest and second-shortest paths, respectively. Each vertex v also has a Boolean flag $deleted(v)$ that becomes true the first time it is deleted from the priority queue. Initially all tentative distances are infinity except $d1(s) = 0$, all parents are null, all flags are false, and the priority queue contains only s. The key of a vertex v in the priority queue is $d1(v)$ if $deleted(v) = false$, $d2(v)$ if $deleted(v) =$ true.

   The algorithm is as follows:

```
while PQ is not empty do
{delete a vertex v of minimum key from PQ
     if deleted(v) = false then
     {   [d1(v) is the key of v and is the shortest distance to v]
          for each w such that (v, w) is an arc do
               if d1(v) + c(v, w) < d1(w) then
               {     d1(w) = d1(v) + c(v, w)
                      p1(w) = v
                      if w is not in PQ then insert w into PQ
                      else decrease-key(w, PQ)
                        [this operation restores PQ following the
                         decrease in the key of w]}
               else if d1(v) + c(v, w) > d1(w) and d1(v) + c(v, w) < d2(w) then
               {     d2(w) = d1(v) + c(v, w)
                      p2(w) = v
                      if w is not in PQ then insert w into PQ
                      else decrease-key(w, PQ)}
      deleted(v) = true
      if d2(v) < infinity then insert v into PQ}
else
{    [d2(v) is the key of v and is the second-shortest distance to v]
      for each w such that (v, w) is an arc do
           if d2(v) + c(v, w) < d2(v, w) then
          {     d2(w) = d1(v) + c(v, w)
                p2(w) = v
                 if w is not in PQ then insert w into PQ
                 else decrease-key(w, PQ)}}
```

Once the while loop finishes, one can trace back the second shortest path from s to v by following $p2$ pointers back from v as long as $d2(p2(x)) + c(p2(x), x) = d2(x)$, where x is the current vertex (initially v) and then following $p1$ pointers until reaching s. The representation of the second shortest paths thus consists of the shortest path tree and a second tree, parts of which can be shared with the shortest path tree. Each second-shortest path consists of a path in the shortest path tree followed by a path in the second-shortest path tree.

The algorithm above runs in $E\log V$ time with a simple PQ implementation or in $E + V\log V$ time with a sophisticated PQ implementation such as a Fibonacci heap. Thus it is much faster than required.

Note that second-shortest paths need *not* be simple; that is, they can contain repeated vertices (Give an example.) If we want second-shortest *simple* paths, then the algorithm above fails. (Give a counterexample.) Instead, we can use the following algorithm, which formed the basis of many student solutions but does not solve the original problem as stated. (The problem statement does not exclude simple paths.)

The idea is to compute a shortest path tree, say by Dijkstra's algorithm, and then delete each arc of this tree from the graph in turn, compute shortest paths from s in the original graph with one arc

deleted, and keep track of the shortest of these paths to each vertex that is longer than the original shortest path. This takes V runs of Dijkstra's algorithm and hence runs in $VE\log V$ time with a simple PQ implementation or in $VE + V^2\log V$ time with a Fibonacci heap or similarly fast PQ implementation. (Give an example of a graph on which this algorithm does not compute second-shortest paths if non-simple paths are allowed.)

b. There are many approaches. The conceptually simplest approach is to run BFS, but maintain an `evenQueue` and an `oddQueue` and give each vertex an even and odd marker. We start by marking the source as even and enqueuing it on the `evenQueue`. From there, we dequeue each vertex v from the `evenQueue` (only the source on the first iteration); we then mark each of v's neighbors as odd and enqueue them on the `oddQueue`, unless they have already been marked in which case we do nothing with v. We next do the same thing with the `oddQueue`, enqueueing their neighbors as even if not marked as even, and alternate between the two queues until both are empty.

   Each vertex is enqueued and edge is used at most twice, so the algorithm is E+V. There are many alternate formulations of this same algorithm.

   By far the most common sub-E+V solution was to only mark odd-distance vertices (or equivalent). This algorithm is correct, but results in V^2 performance for certain graph topologies, e.g. the one shown below. If we generalize this graph (by adding additional middle vertices), then the vertices corresponding to A/B/C/D will be enqueued a total of roughly V/2 times. This results in V^2 runtime.