COS 226 **Algorithms and Data Structures** **Spring 2024**

Final Solutions

1. **Initialization.**

   Don't forget to do this.

2. **Graph search algorithms.**

   (a) 0 2 6 8 3 5 1 4 7 9

   (b) 0 2 8 3 1 5 9 7 4 6

   (c) 5 7 9 1 6 4 3 8 2 0

   (d) no
   The digraph is not a DAG. For example, $3 \rightarrow 1 \rightarrow 9$ *is a directed cycle.*

3. **Minimum spanning trees.**

   (a) 10 20 30 40 50 90 120

   (b) 90 10 50 20 40 30 120

4. **Shortest paths.**

   (a) 0 4 5 3 1 2
   (b) 0 80 100 70 30 40
   (c) 0 1 3 4 5
   Compare the `distTo[]` *values computed in part (b) to the ones obtained by the execution of Kruskal in part (a) (the latter are optimal).*

5. **Maxflows and mincuts.**

   (a) 36 = 16 + 2 + 25 - 7

   (b) 107 = 28 + 10 + 30 + 30

   (c) $A \rightarrow F \rightarrow G \rightarrow B \rightarrow H \rightarrow D \rightarrow I \rightarrow J$

   (d) 71 = 28 + 7 + 36

   (e) A F G or A B C F G

6. **Data structures.**

   (a) T T F F
   Insert each option to an empty hash table.

   (b) (20, 4), (17, 8)
   The constraints of the 2d-tree imply that, for any point (x, y) in T, we must have both $x \geq 12$ and $3 < y \leq 10$.

7. **Properties of graph algorithms.**

   (a) T: Let (v, u) be the lightest edge in the graph. Consider the cut with v on one side and all other vertices on the other. Since (v, u) is the lightest edge crossing this cut, it must appear in every MST.

   (b) F: Consider a graph with three vertices: s, t, and v, where the edge $s \to t$ has a weight of 3, and the edges $s \to v$ and $v \to t$ each have a weight of 2. Initially, the shortest path from s to t is $s \to t$ with a length of 3. However, after squaring the edge weights, the shortest path becomes $s \to v \to t$, with a total length of $4 + 4 = 8$.

   (c) T: The value of a minimum st-cut is equal to the value of a maximum flow. Given a minimum cut in the graph (represented as a set of vertices S on s's side of the cut), the value of the cut can be calculated by iterating over all edges and summing the capacities of edges (u, v) where $u \in S$ and $v \notin S$.

   (d) F: The two augmenting paths can share overlapping edges. For instance, consider a zero-flow in a flow network with four vertices s, t, v, and u, and the following edges: $s \to v$ with a capacity of 10, $v \to u$ with a capacity of 5, $u \to t$ with a capacity of 5, and $v \to t$ with a capacity of 10. This flow has two augmenting paths: $s \to v \to t$ with a bottleneck capacity of 10, and $s \to v \to u \to t$ with a bottleneck capacity of 5. Despite these paths, the maximum flow value is only 10.

   (e) T: Let e_1 and e_2 be the edges crossing the mincut. Removing these edges breaks the graph into exactly two connected components, with vertex sets V' and $V \setminus V'$. When Kruskal's algorithm is run by Karger's algorithms, it processes edges in ascending order of weight. Since e_1 and e_2 are the heaviest edges, the partial MST will reduce to two connected components before processing e_1 and e_2. These components must correspond to the vertex sets V' and $V \setminus V'$, as e_1 and e_2 are the only edges connecting V' and $V \setminus V'$ and have not yet been processed.

8. **Dynamic programming.**

   A K D E I

```
int opt[] = new int[n + 1];
for (int i = 0; i <= n; i++) {
    opt[i] = values[i];
    for (int j = 1; j < i; j++)
        opt[i] = Math.max(opt[i], values[j] + opt[i-j] - 1);
}
return opt[n];
```

9. **Randomness.**

   T T F F T

   (a) T: `findOneA` uses randomness while `findOneB` does not.
   (b) T: The loop in `findOneA` runs $0.1n$ times, leading to a $\Theta(n)$ worst-case running time. The expected running time is constant because each iteration of the loop takes constant time, and the algorithm halts after each iteration with constant probability (in this case, with proability 0.1).
   (c) F: If the first $0.9n$ entries of `a` are zeros, `findOneB` will check $\sim 0.9n$ entries.
   (d) F: If all the sampled indices are the same, for example, index 0, and `a[0] = 0`, the algorithm makes a mistake.
   (e) T: The array `a` contains entries with the value one, and since `findOneA` may scan the entire array, it is guaranteed to find and return one of these entries.

10. **Multiplicative weights.**

    F F F T T

    (a) F: If the first expert predicts incorrectly on each of the last 10 days and the second expert predicts incorrectly on the first 5 days, the second expert is eliminated on the first day, while the first expert is eliminated on day $T-9$.
    (b) F: If all the experts make incorrect predictions every day, the suggested algorithm will always predict correctly.
    (c) F: The weight of the correct experts remains 1 throughout the algorithm's execution. After the first day, the total weight of the incorrect experts becomes $\frac{n-1}{2}$. After the second day, it reduces to $\frac{n-1}{4}$, and so on. Thus, it takes $\Theta(\log n)$ days for the weight of the correct experts to surpass or equal the total weight of the incorrect experts.
    (d) T: Repeat the analysis of the multiplicative weight algorithm. This time, the total weight of the best experts is $\frac{n}{4} \cdot (\frac{1}{2})^M$, yielding the inequality $\frac{n}{4} \cdot (\frac{1}{2})^M \le (\frac{3}{4})^m \cdot n$. Observe that n cancels.
    (e) T: The weight of the first point is 2^5 times that of the second point, indicating its weight was doubled 5 more times than the second's. With only 5 iterations, the first point's weight was doubled in every iteration. Since a point's weight doubles only when a decision stump mislabels it, this means the first point was mislabeled by all decision stumps.

11. **Intractability.**

    T F T T T T

    (a) T: Since X is **NP**-complete, it is in **NP**. Since Y is **NP**-complete, every problem Z in **NP** poly-time reduces to Y. In particular, X poly-time reduces to Y. A similar argument shows that Y poly-time reduces to X.
    (b) F: If X poly-time reduces to Y, a polynomial-time algorithm for Y can be used to construct a polynomial-time algorithm for X, but the reverse is not necessarily true.

(c) T: Since Y poly-time reduces to Z, a $T(n)$-time algorithm for Z can be used to construct a $T(p(n))$-time algorithm for Y, where $p(n)$ is some polynomial. Similarly, since X poly-time reduces to Y, a $T(p(n))$-time algorithm for Y can be used to construct a $T(p(q(n)))$-time algorithm for X, where $q(n)$ is another polynomial. Therefore, a $T(n)$-time algorithm for Z implies a $T(f(n))$-time algorithm for X, where $f(n) = p(q(n))$ is a polynomial.

(d) T: If **P** $\neq$ **NP**, then there exists a problem X in **NP** that cannot be solved in polynomial time. Since **SAT** is **NP**-complete, X poly-time reduces to **SAT**. Given that X does not have a polynomial-time algorithm and poly-time reduces to **SAT**, it follows that **SAT** also cannot be solved in polynomial time.

(e) T: Every `YES` instance of the problem has a witness, and the verification algorithm correctly validates the witness in polynomial time.

12. **Design: shortest paths through a landmark.**

(a) **Constructor:** Run Dijkstra's algorithm twice: once using s as the source vertex, and once using x as the source vertex. Store an integer variable `sToX` containing the distance from s to x (which is contained in `distTo[x]` of the Dijkstra's run that used s as the source) and also the `distTo` array of the Dijsktra's run that used x as a source, both as instance variables.

pathLen: Report the sum of the distance from s to x and the distance from x to v, i.e. `sToX + distTo[v]`.

(b) **Constructor:** First, run Dijsktra's in G using x as the source vertex and store the `distTo` array as an instance variable.

Compute the reverse graph of G (obtained by reversing each edge in G and keeping the same vertices), call it G'. Run Dijkstra's in G' using x as the source and store the `distTo` array as an instance variable called `distToReverse`. The value of `distToReverse[u]` corresponds to the shortest path from u to x for any u.

pathLen: Report the sum `distTo[v] + distToReverse[s]`.

13. **Design: shortest path with a reverse edge.**

(a) No, the simplest example is a graph with two vertices s and t and one directed edge from s to t.

(b) Construct a new graph G'. Create two copies of G and add them to G', call the first copy G_0 and the second copy G_1. For every edge $(u, v) \in G$, add an edge from v_0 to u_1 in G', i.e. an edge from the copy of vertex v in G_0 to the copy of vertex u in G_1. To find the shortest almost-path, run a BFS from s_0 and report the distance to t_1 (so from the copy of vertex s in G_0 to the copy of vertex t in G_1).

The key idea of this solution is that vertices in G_0 correspond to paths only taking edges in the normal direction, and vertices in G_1 correspond paths that have taken exactly one edge in the opposite direction (and that's why why add an egde from v_0 to u_1 for each edge (u, v), note how the vertices are switched).

(c) Alter the full solution to add a "super sink", i.e. a new vertex t' and edges from t_0 and t_1 to t'. Run BFS to find shortest path from s_0 to t' and report the result minus 1.

Alternate solution: Run a BFS on the graph G from s to t, and report the minimum between this and the result found by the solution in part b.