COS 226 — Algorithms and Data Structures — Fall 2020

Final Exam Solutions

1. **Initialization.** Don't forget to do this.

2. **Memory usage.**

   (a) 3,276,800 (100×2^{15})

   The running time for $V = E = 80$ is $102400 = 100 \times 2^{10}$. Going down one row makes the running time go up by a factor of 2; going right one column makes the running time go up by a factor of 8. So, the running time for $V = 320$ and $E = 160$ is $100 \times 2^{10} \times 2^2 \times 8 = 100 \times 2^{15}$.

   (b) $\Theta(VE^3)$

   When you double V, the running time goes up by a factor of 2, so the exponent for V is $\log_2 2 = 1$. When you double E, the running time goes up by a factor of 8, so the exponent for E is $\log_2 8 = 3$.

3. **String operations.**

   (3.1) n^2

   String concatenation takes time proportional to the length of the resulting string, so the running time is $\Theta(1 + 2 + 3 + \ldots + n) = \Theta(n^2)$.

   (3.2) n

   Appending each character to the end of a `StringBuilder` *object takes $\Theta(1)$ amortized time. So, starting from an empty* `StringBuilder`*, appending n characters takes $\Theta(n)$ time in the worst case.*

   (3.3) n^2

   Each recursive call takes $\Theta(n)$ time and reduces the length of the string by 1. So, the running time is $\Theta(1 + 2 + 3 + \ldots + n) = \Theta(n^2)$.

   (3.4) $n \log n$

   As with mergesort, each recursive call takes $\Theta(n)$ time, plus the time for two subproblems of length $n/2$. So, the overall running time is $\Theta(n \log n)$.

4. **String sorts.**

   (4.1) *MSD radix sort (after the second call to key-indexed counting)*

   (4.2) *LSD radix sort (after 2 passes)*

   (4.3) *LSD radix sort (after 1 pass)*

   (4.4) *3-way radix quicksort (after the first partitioning step)*

   (4.5) *MSD radix sort (after the first call to key-indexed counting)*

5. **Graph search.**

   (5.1) 4 0 1 2 7 3 6 8 9 5

   (5.2) 7 2 6 3 8 1 9 0 5 4

   (5.3) 4 0 3 5 6 1 8 9 2 7

6. **Minimum spanning trees.**

   (6.1) 10 20 30 50 60 100 120

   (6.2) 30 10 50 20 60 100 120

   (6.3) 70
   If the weight is 70, edge r–s will be an edge in one of the two MSTs of G.
   If the weight is 69 or less, edge r–s will be an edge in the unique MST of G.

7. **Maximum flow.**

   (7.1) $61 = 5 + 30 + 26$

   (7.2) 35
   Flow conservation implies that the flow into vertex C equals the flow out of vertex C. The flow into C is $15 + 15 + 3 + 2 = 35$. The only edge leaving C is $C \to D$.

   (7.3) $63 = 5 + 30 + 28$

   (7.4) $A \to F \to G \to B \to C \to I \to J$

   (7.5) $2 = \min\{4, 3, 10, 10, 2, 7\}$

   (7.6) 63.
   The value of the flow after sending 2 units of flow along the augmenting path identified in (7.4) is $61 + 2 = 63$. Since this equals the capacity of the cut in (7.3), it is a maxflow.

8. **Key-indexed counting.**

   (8.1) the integer $r - 1$

   (8.2) less than r

   (8.3) less than or equal to r

   (8.4) would make key-indexed counting unstable

   (8.5) none of the above

9. **Data compression.**

   (9.1) 5 G I T E R S

   (9.2) C C B A D B C

   (9.3) B C B C C B B C C A

10. **Ternary search tries.**

    (10.1) N O P Q

    (10.2) WE

11. **Properties of graph algorithms.**

    (11.1) MST
    All spanning trees for a graph with V vertices have $V-1$ edges. So, this transformation will increase the weight of all spanning trees by the same amount.

    (11.2) Shortest path, Longest path, MST, Mincut
    Multiplying the edge weights (or capacities) by a constant is equivalent to changing the units (e.g., from inches to feet) and does not affect the solution to the problem.

12. **Properties of string algorithms.**

    (12.1) all three options

    (12.2) first two options only

    (12.3) first and last options only

13. **Problem identification.**

    (13.1) Possible
    Run DFS (or BFS) from any vertex s and check that all vertices get marked.

    (13.2) Possible
    Run DFS (or BFS) from any vertex s and check that all vertices are marked. Then, run DFS (or BFS) from s in the reverse digraph and check that all vertices are marked.

    (13.3) Possible
    Negate the weights and run Kruskal or Prim. Both algorithms work with negative edge weights.

    (13.4) Impossible
    It takes at least $\Theta(V^2)$ time to initialize the V^2 array entries. This can be larger than $\Theta(E \log E)$.

    (13.5) Possible
    LSD (or MSD) radix sort the integers and check adjacent entries for the closest pair.

    (13.6) Possible
    MSD radix sort the strings. Check adjacent entries for equal strings.

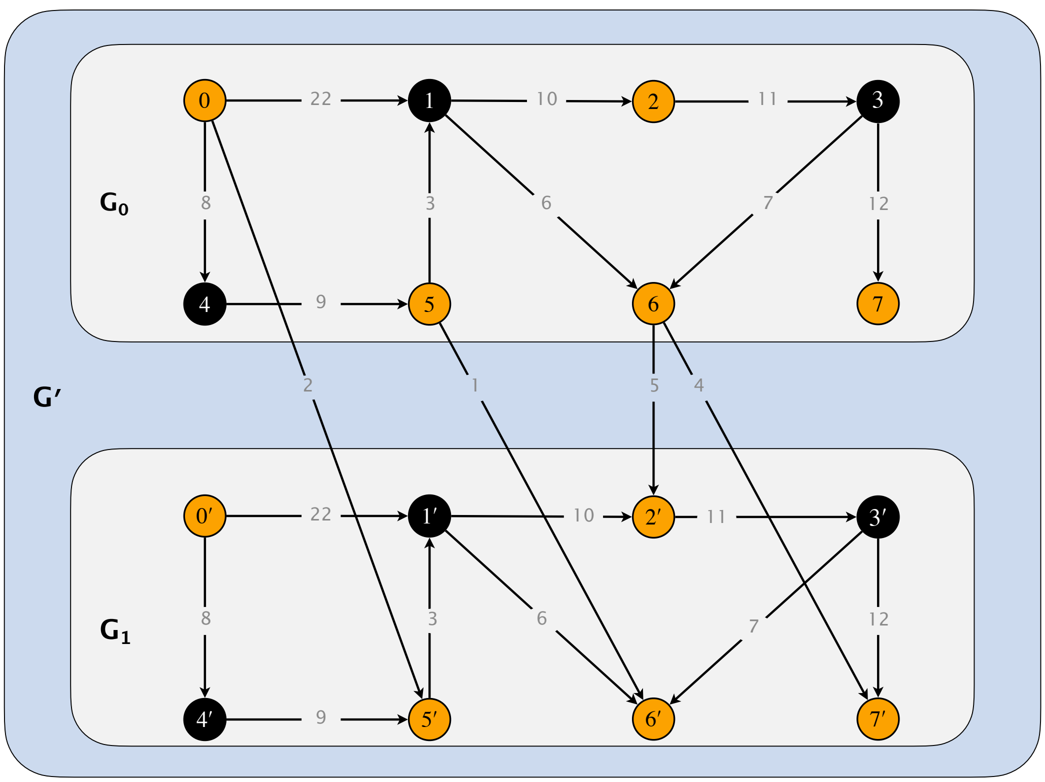
    (13.7) Impossible
    Huffman codes are optimal in the sense that no prefix-free code can use fewer bits for a given message.

14. **Shortest almost-alternating path.**

The key idea is to create two "copies" of G: G_0 and G_1. Paths from the source to vertices in G_0 alternate colors at every vertex. Path from the source to vertices in G_1 alternate colors at every vertex except one—the edge that makes the leap from G_0 to G_1.

- For each vertex v in G, add two vertices v_0 and v_1 to G'. The source vertex s' is s_0. Interpretation: paths from s' to v_0 correspond to paths in G from s to v that alternate colors at every vertex; paths from s' to v_1 correspond to paths in G from s to v that alternate at every vertex except one.
-
  - for each edge (v, w) in G with v and w of opposite colors: add the two edges (v_0, w_0) and (v_1, w_1) to G':
  - for each edge (v, w) in G with v and w of the same color: add the edge (v_0, w_1) to G'.
- The shortest almost-alternating path corresponds to either the shortest path from s_0 to t_0 (alternates colors at every vertex) or from s_0 to t_1 (alternates colors at every vertex except one).

For illustration, here is the digraph G' corresponding to the digraph on the exam.



15. **Wildcard match.**

The main idea is to adapt a 4-way trie.

- *Adding a string is exactly the same.*
- *Wildcard match is similar to search in a trie except that when you encounter the wildcard character, you go down each of the four branches.*

Here is a complete Java implementation.

```
public class Wildcard {
    private Node root;

    private static class Node {
        private boolean marked;
        private Node a, c, t, g;
    }

    public void add(String s) {
        root = add(root, s, 0);
    }

    private Node add(Node x, String s, int d) {
        if (x == null) x = new Node();
        if (d == s.length()) {
            x.marked = true;
            return x;
        }
        char c = s.charAt(d);
        if      (c == 'A') x.a = add(x.a, s, d+1);
        else if (c == 'C') x.c = add(x.c, s, d+1);
        else if (c == 'G') x.g = add(x.g, s, d+1);
        else if (c == 'T') x.t = add(x.t, s, d+1);
        else throw new IllegalArgumentException();
        return x;
    }

    public boolean matches(String t) {
        return matches(root, t, 0);
    }

    private boolean matches(Node x, String t, int d) {
        if (x == null) return false;
        if (d == t.length()) return x.marked;
        char c = t.charAt(d);
        if      (c == 'A') return matches(x.a, t, d+1);
        else if (c == 'C') return matches(x.c, t, d+1);
        else if (c == 'G') return matches(x.g, t, d+1);
        else if (c == 'T') return matches(x.t, t, d+1);
        else if (c == '.') return matches(x.a, t, d+1) ||  matches(x.c, t, d+1)
                               || matches(x.g, t, d+1) ||  matches(x.t, t, d+1);
        else throw new IllegalArgumentException();
    }
}
```