COS 226 Algorithms and Data Structures Fall 2019

Final Exam Solutions

0. **Initialization.**

   Put your name, NetID, and precept number on your cheatsheet!

1. **Empirical running time.**

   (a) 3200

   (b) $VE^{1/2}$

2. **Memory.**

   $64 + 8r$

   - 16 bytes of object overhead
   - 8 bytes of inner class overhead
   - 4 bytes for int
   - 4 bytes of padding
   - 8 bytes for reference to array
   - $24 + 8r$ bytes for array of references

3. **String sorts.**

   A C D B B C E

   - A: original input
   - C: MSD radix sort after the first call to key-indexed counting
   - D: 3-way radix quicksort after the first partitioning step
   - B: LSD radix sort after 1 pass
   - B: LSD radix sort after 2 passes
   - C: MSD radix sort after the second call to key-indexed counting
   - E: Sorted

4. **Depth-first search.**

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

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

   (c) $1 \to 0$

   A digraph has a topological order if and only if it has no directed cycles. In this digraph, all directed cycles use edge $1 \to 0$.

5. **Breadth-first search.**

   (a)

   ■ It terminates.

   ■ Some vertices are added to the queue more than once.

   ■ At some point, the queue contains multiple copies of the same vertex.

   □ Upon termination, `distTo[1]` is `1` (the length of the *shortest* path from 0 to 1).

   ■ Upon termination, `distTo[9]` is `9` (the length of the *longest* path from 0 to 9).

   Upon termination, `distTo[v]` *is the length of the longest path from* s *to* v *for each* v*. This property holds for any DAG (but we note that the code would go into an infinite loop if run on a digraph with a directed cycle reachable from* s*).*

   (b) Need to mark the vertices when they are added to the queue, both for correctness and efficiency.

   - Initialize `marked[s] = true` before `while` loop.
   - Set `marked[w] = true` in the body of `if` statement.
   - Optionally, can set `distTo[v]` to infinity or `Integer.MAX_VALUE` (provided `distTo[s]` is still initialized to 0).

   Setting `marked[v] = true` *when dequeueing* `v` *is too late: this will result in some vertices being on the queue more than once.*

6. **Minimum spanning tree.**

   (a) 10 20 30 60 70 100 120

   (b) 60 70 20 10 30 100 120

7. **Knuth–Morris–Pratt substring search.**

(a)

	0	1	2	3	4	5	6	7
`A`	0	0	0	0	0	0	0	8
`B`	1	2	2	4	5	6	2	4
`C`	0	0	3	0	0	3	7	0

4

8. **Java String library performance.**

A A E D G I E H

The surprisingly slow worst-case running times for `indexOf()` *and* `matches()` *were discussed in iClicker questions during lecture.*

9. **Burrows–Wheeler data compression.**

   A A E G H

   (a) Move-to-front, on its own, does not do any compression. If the input has n bytes, the move-to-front encoded input will also have n bytes. So, the compression ratio is exactly 1.

   (b) The Burrows–Wheeler transform, on its own, does not do any compression. In fact, it adds one 32-bit integer to the result. That is, if the input has n bytes, the Burrows–Wheeler transform will have $n + 4$ bytes. The compression ratio is ~ 1.

   (c) The three characters (X, Y, and Z) appear with equal frequency. Thus, the Huffman code will use 1 bit for one of the characters and 2 bits for the other two. Thus, for every three 8-bit input characters (24 bits), the correspond part of the Huffman-encoded message will use 5 bits. The overhead for encoding the Huffman trie is negligible (as a function of n). The compression ratio is ~ 5/24.

   (d) The result of move-to-front is the following sequence of bytes (where `58` is `X`, `59` is `Y`, and `5A` is `Z`):

   ```
   58  0 59  0 5A  0  2  0  2  0  2  0  2  0  2  0  2  0  2  0  2  0  2  0  2  0  2  0  2  0
   ```

   Ignoring the first few entries, the values alternate between 0 and 2. As a result, the Huffman trie encodes `0` with one bit and `2` with two bits. Thus, for every two 8-bit input characters (16 bits), the corresponding part of the Huffman-encoded message uses 3 bits. The compression ratio is ~ 3/16.

   (e) The result of Burrows–Wheeler transform is the following sequence:

   ```
   0
   Z Z Z Z Z X X X X X X X X X X Y Y Y Y Y Z Z Z Z Z Y Y Y Y Y
   ```

   Ignoring the encoding of the first integer 0 and the newline character, the result of applying move-to-front to this input sequence is

   ```
   5A  0  0  0  0 59  0  0  0  0  0  0  0  0  0 5A  0  0  0  0  2  0  0  0  0  1  0  0  0  0
   ```

   Ignoring a constant number of entries, all values are 0. As a result, the Huffman trie will encode 0 with one bit. Thus, with the exception of a constant number of places, for every 8-bit input character (8 bits), the corresponding part of the Huffman-encoded message uses 1 bit. The compression ratio is ~ 1/8.

10. **Why did we do that?**

    B B A B B A C A

11. **Shortest paths.**

    G J A C E L H L H

```
public BellmanFordSP(EdgeWeightedDigraph G, int s) {

    // initialize data structures
    distTo = new double[G.V()];
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;

    // repeat V-1 times
    for (int i = 1; i < G.V(); i++) {

        // for every edge e = v->w in the digraph
        for (int v = 0; v < G.V(); v++) {
            for (DirectedEdge e : G.adj(v)) {
                int w = e.to();

                // relax edge e = v->w
                if (distTo[w] > distTo[v] + e.weight())
                    distTo[w] = distTo[v] + e.weight();

            }
        }
    }
}
```

12. **Ternary search tries.**

    yes: DATA DO HUE PRO QUEUE TRIE

    no: BRIE DAD DARK FUN TRUE

13. **Regular expressions.**

    (a) 1→4, 1→7, 3→6, 7→1, 8→9, 9→8

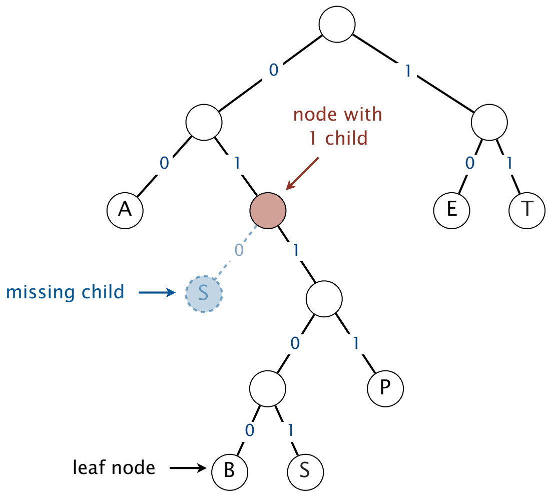
    (b) delete edge 7→1

14. **Prefix-free codes. (10 points)**

   (a) 010

   None of the known codewords are prefixes of 010 (but some are prefixes of 01).

   (b) The key idea is the correspondence between binary tries and prefix-free codes. In any optimal prefix-free code (such as a Huffman code) every node is either a leaf node (containing a symbol and no children) or an internal node (with two children). If one codeword is missing, there will be exactly one node with a single child. That node's missing child corresponds to the missing codeword.



To deduce the missing codeword:

- insert the $n - 1$ known codewords into a binary trie
- do an inorder traversal of the trie
  - use a *stack* (or `StringBuilder`) to maintain the sequence of bits on the path from the root to the current node
  - when you find the node with exactly one child, use the stack to determine the codeword corresponding to the missing child

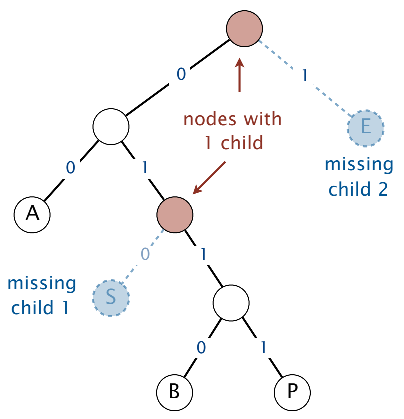
A preorder or postorder traversal of the trie is also fine.

Instead of using a stack to determine the missing codeword, you could store the codewords in the leaf nodes and the corresponding prefixes of the codewords in the internal nodes.

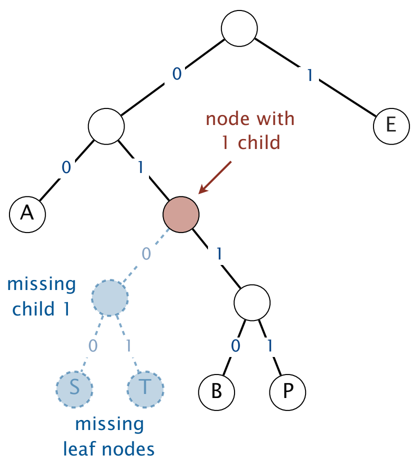
(c) When two codewords are missing, there are two possibilities for the trie that results from inserting the $n - 2$ known codewords:

- *Case 1: two different nodes have exactly one child.* In this case, everything is the same as in (a) except that you will find two nodes with exactly one child (and each missing child corresponds to a different missing codeword).
- *Case 2: only one node has exactly one child.* This case is similar to (a) except that, now, the missing child is an internal node in the original Huffman trie and its two children are leaf nodes (corresponding to the two missing codewords).

You may need to perform two trie traversals: the first traversal determines whether you are in Case 1 or Case 2, and the second traversal deduces the missing codewords.



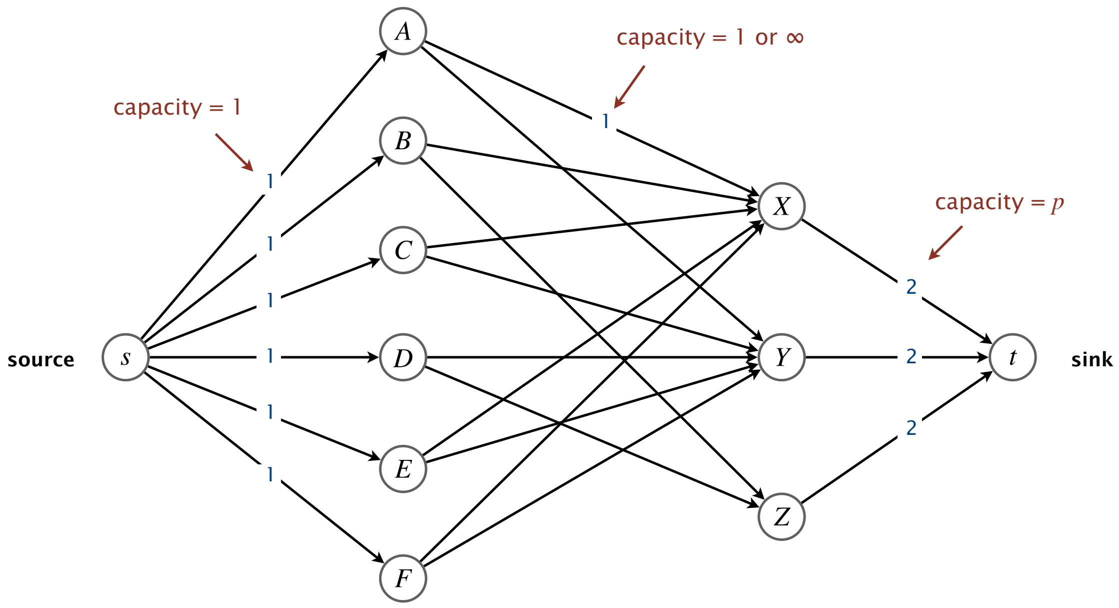
Case 1: two nodes with 1 child



Case 2: one node with 1 child

15. **Writing seminar assignment problem. (10 points)**

(a)



(b) If the value of the maximum flow is n.

(c) n

Each augmenting path increases the value of the flow by exactly 1. So, there will be exactly n augmenting paths in the worst case.

(d) Include an edge from each student to their top k writing seminars (instead of to their top 2). The outdegree of each student vertex will be k (instead of 2).

(e) Here are two efficient solutions.

Solution 1: binary search. If $n > pm$, then it's not possible to assign the students to writing seminars because there are more students than total seminar capacity.

Otherwise, use *binary search* to find k^*. Specifically, maintain an interval $[lo, hi]$ so that when $k \leq lo$, there is no assignment (of students to seminars in which each students gets one of their top k choices); and when $k \geq hi$, there is such an assignment.

- Initialize $[lo, hi] = [0, m]$.
- Repeat until $hi - lo = 1$:
  - Chose $k = \lfloor (lo + hi)/2 \rfloor$ and solve the corresponding maxflow problem from (d).
  - If there is an assignment, update $hi = k$; otherwise update $lo = k$.
- Return $k^* = hi$.

In the worst case, this reduces the number of maxflow problems we need to solve from m to $\log_2 m$.

Solution 2: incremental maxflow. The naive approach is to solve the assignment problem in (d) for each $k = 1, 2, \ldots, m$ until there is an assignment. In the worst case, this requires solving m maxflow problems from scratch.

However, the maxflow problems for k and $k + 1$ are remarkably similar, so you can reuse the work done in solving one problem to jumpstart the next. Specifically, after computing the maximum flow f^* corresponding to a given value of k:

- Add an edge from each student to their $k + 1^{st}$ top choice.
- Solve a maxflow in the updated network, but use f^* as the initial flow (instead of the zero flow).

Now, the total number of augmenting paths is n, which is the same, in the worst case, as the number of augmenting paths to find an assignment for one specific value of k (starting from scratch)!