Homework 9

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

November 10, 2017

Problem 1. Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

Solution.

Proof by Method of Contradiction
Assumptions:

- A binary tree corresponding to an optimal prefix code, say T.
- Assume that T is not full.

Let node p have a single child x. Let T' be the tree obtained by removing p and replacing it by x. Let q be a leaf node which is a descendant of x. Then we have:

$$cost(T') \leq \sum_{c \in C \setminus \{m\}} c.freq.d_T(c) + m.freq(d_T(m) - 1) < \sum_{c \in C} c.freq.d_T(c) = cost(T)$$

which contradicts the fact that T was optimal. Therefore every binary tree corresponding to an optimal prefix code is full. □

Problem 2. Construct an optimal Huffman code (by showing the tree) for the following set of alphabets and frequencies: `a:32 b:5 c:12 d:10 e:6 f:15 g:32 h:21`

Solution:

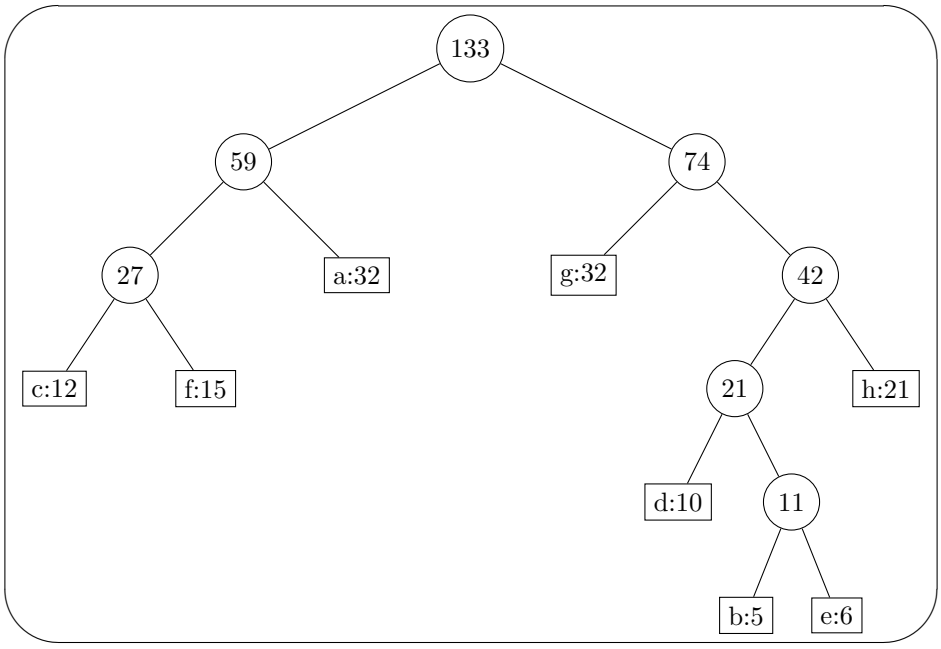

Fig 6.1: Binary Tree for optimal Huffman code.

Every left child denotes a corresponding 0 in the Huffman code and every right child denotes a 1.

□

Problem 3a. Design $O(n)$ algorithm to test if a given undirected graph G on n nodes is acyclic. Notice, the running time of your algorithm should not depend on the number of edges m!

Solution:

An undirected graph is acyclic (i.e., a forest) iff a DFS yields no back edges. Since back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree, so no back edges means there are only tree edges, so there is no cycle.

So we can simply run DFS. If find a back edge, there is a cycle. The complexity is $O(n)$ instead of $O(m+n)$. Since if there is a back edge, it must be found before seeing n distinct edges. This is because in a acyclic undirected graph (forest),

$$|E| \leq |V| + 1 \implies m \leq n + 1$$

Pseudo-Code:

```
def DFSCheckCycle(G):
    n = len(G) # Number of vertices
    p = [None]*(n) #parent array

    color = ["white"]*(n)
    cycle = False

    for i in range(n):
        if color[i] == "white":
            p[i] = -1 # meaning it is a root of a DFS-tree of the DFS forest
            cycle = visit(i, G, color)
            if cycle:
                return cycle

    return False

def visit(u, G, color):
    cycle = False
    color[u] = "gray"
    for v in G[u]:
        if color[v] == "white":
            p[v] = u
            if visit(v, G, color):
                return True
        elif color[v] == "gray":
            cycle = True
            break
    color[u] = "black"# once DFS for this vertex ends, assign its color to black

    return cycle

# Sample Graph (using Adj List)
G = {0: [1],
    1: [2],
```

```
35      2: [1,3],
36      3: [2]}
```

□

Problem 3b. Extend the above algorithm to actually print the cycle, in case G is cyclic.

Solution:

Pseudo-Code:

```
1  def DFSCheckCycle(G):
2      n = len(G) # Number of vertices
3      p = [None]*(n) #parent array
4
5      color = ["white"]*(n)
6
7      for i in range(n):
8          if color[i] == "white":
9              p[i] = -1 # meaning it is a root of a DFS-tree of the DFS forest
10             visit(i, G, color)
11
12 def visit(u, G, color):
13     cycle = False
14     color[u] = "gray"
15     for v in G[u]:
16         if color[v] == "white":
17             p[v] = u
18             visit(v, G, color)
19         elif color[v] == "gray":
20             cycle = True
21             break
22     color[u] = "black"# // once DFS for this vertex ends, assign its color to black
23
24     if cycle:
25         printCycle(v,u)
26
27 def printCycle(v, u):
28     print(u)
29     u = p[u]
30
31     while u != v:
32         print(u)
33         u = p[u]
34
35 # Sample Graph (using Adj List)
36 G = {0: [1],
37     1: [2,3],
38     2: [1,3],
39     3: [1,2]}
```

□

Problem 4. You have an undirected graph $G = (V, E)$ and two special nodes $r, d \in V$. At time 0, node r is republican, node d is democratic, while all the other nodes $v \notin \{r, d\}$ are initially undecided. For every

$i = 1, 2, 3, ...$, the following 2-stage conversion process is performed at time time i. At the first stage, all republicans at time $(i1)$ look at all their neighboring nodes v which are still undecided, and convert those undecided nodes to become republican. Similarly, at the second stage, all democratic nodes at time $(i1)$ look at all their neighboring nodes v which are still undecided by the end of the first stage above, and convert those undecided nodes to become democratic. The process is repeated until no new conversions can be made. For example, if G is a 5-cycle 1, 2, 3, 4, 5 where $r = 1$, $d = 5$, after time 1 node 2 becomes republican and node 4 becomes democratic, and after time 2 the last remaining node 3 becomes republican (as republicans move first). On the other hand, if the initial democratic node was d = 3 instead, then already after step 1 nodes 2 and 5 become republican, and node 4 becomes democratic, and no step 2 is needed.

Assume each node v have a field v.color, where red means republican, blue means democratic, and white means undecided, so that, at time 0, $r.color = red$, $d.color = blue$, and all other nodes v have $v.color = white$.

Problem 4a. Give a dynamic-programming-style recurrence equation for the minimum number of stops you need to make. Briefly justify the correctness of this equation.

Solution:

Pseudo-Code:

```
1  import sys
2  def BFS(G, s):
3      for u in range(len(G)):
4          if u == s:
5              pass
6
7      p = [None]*(n) #parent array
8      color = ["white"]*(n)
9      d = [sys.maxsize]*(n)
10
11     color[s] = "gray"
12     d[s] = 0
13
14     q = []
15     q.insert(0,s)
16
17     while len(q):
18         u = q[-1]
19         del q[-1]
20
21         for v in G[u]:
22             if color[v] == "white":
23                 color[v] = "gray"
24                 d[v] += 1
25                 p[v] = u
26                 q.insert(0,s)
27
28         color[u] = "black"
29     return d
30
31 def politics(G, r, d):
32     d1 = BFS(G, r)
33     d2 = BFS(G, d)
34
35     color = ["white"]*(n)
36
```

```
    for i in len(G):
        if d1[i] > d2[i] :
            color[i] = "red"
        elif d1[i] == d2[i] and d1[i] != sys.maxsize:
            color[i] = "red"
        elif d1[i] < d2[i]:
            color[i] = "blue"
```

□

Problem 4b. Show how speed up your procedure in part (a) by a factor of 2 (or more, depending on your implementation) by directly modifying the BFS procedure given in the book. Namely, you must compute the final colors of each node by performing a single, appropriately modified, BFS traversal of G. Please write pseudocode similar to the standard BFS pseudocode, and briefly explain your code (no proof needed).

Solution:

Using a queue, we just insert the starting nodes in the required order and colorise nodes according to their parent.

Pseudo-Code:

```
import sys
def BFS_mod(G, r, d):
    for u in range(len(G)):
        if u == s:
            pass

    p = [None]*(n) #parent array
    color = ["white"]*(n)
    d1 = [sys.maxsize]*(n)
    d2 = [sys.maxsize]*(n)

    color[r] = "red"
    color[d] = "blue"
    d1[r] = 0
    d2[d] = 0

    q = []
    q.insert(0,r)
    q.insert(0,d)

    while len(q):
        u = q[-1]
        del q[-1]

        for v in G[u]:
            if color[v] == "white":
                color[v] = color[u]
                d[v] += 1
                p[v] = u
                q.insert(0,s)
```

Since there is going to be just a single BFS call, and hence all the vertices are going to be visited just once, there is speed up of 2

□

Problem 4c. Now assume that at time 0 more than one node could be republican or democratic. Namely, you are given as inputs some disjoint subsets R and D of V , where nodes in R are initially republican and nodes in D are initially democratic, but otherwise the conversion process is the same. For concreteness, assume $|R| = |D| = t$ for some $t \geq 1$ (so that parts (a) and (b) correspond to t = 1). Show how to generalize your solutions in parts (a) and (b) to this more general setting. Given parts (a) and (b) took time $O(|V| + |E|)$ (with different constants), how long would their modifications take as a function of $t, |V|, |E|$? Which procedure gives a faster solution?

Solution: For part (a), we modify it by doing BFS starting from all the R nodes, getting respective distances and taking the minimum for each node. Similarly, we do it for D nodes and decide the nodes color using the minimum starting in from them.

For part (b), we simply enqueue all the R nodes and then enqueue the D nodes and the remaining algorithm would be the same.

Since in the improved part (a), we will do the same process t times, we will get $O((E + V)t)$ running time.

In the improved part (b), the only extra step is to enqueue all current R and D nodes, so complexity will be O(V + E + t). Hence the second approach is a faster solution. □