CS180 HW4

Siddharth Joshi

TOTAL POINTS

94 / 100

QUESTION 1

1 Problem 1 **20 / 20**

- ✓ **- 0 pts** Correct
- **- 3 pts** did not update queue
- **- 7 pts** Huffman coding not correct
- **- 10 pts** algorithm runtime not correct
- **- 15 pts** wrong answer but showed efforts
- **- 20 pts** no answer

QUESTION 2

2 Problem 2 **20 / 20**

- ✓ **- 0 pts** Correct
- **- 4 pts** did not update the index
- **- 10 pts** algorithm runtime not correct
- **- 15 pts** wrong answer but showed efforts
- **- 20 pts** no answer

QUESTION 3

Problem 3 35 pts

3.1 3.a **5 / 5**

- ✓ **- 0 pts** Correct
- **- 2 pts** Need more explanation
- **- 5 pts** No answer found

3.2 3.b **10 / 10**

- ✓ **- 0 pts** Correct
- **- 5 pts** Click here to replace this description.
- **- 3 pts** Need more explanation
- **- 10 pts** No answer found

3.3 3.c **7 / 10**

- **- 0 pts** Correct
- ✓ **- 3 pts** Need more explanation
- **- 5 pts** Need more explanation
- **- 10 pts** No answer found

3.4 3.d **7 / 10**

- **- 0 pts** Correct
- ✓ **- 3 pts** Need more explanation
- **- 5 pts** Need more explanation
- **- 10 pts** No answer found

QUESTION 4

Problem 4 25 pts

4.1 4.a **15 / 15**

- ✓ **- 0 pts** Correct
- **- 15 pts** No answer found
- **- 5 pts** Lack step-to-step description of algorithm
- **- 10 pts** wrong answer but showed efforts

4.2 4.b **10 / 10**

- ✓ **- 0 pts** Correct
- **- 3 pts** Minor mistake
- **- 5 pts** Lack step-to-step description of algorithm
- **- 10 pts** No answer found

HW4

1. ↓ this list is called access

CODE (LIST OF N FILES where the index is the 'name' of the file and the value at an index is the # of times that file is accessed):

→ CREATE N ~~NODES~~ leaf NODES (1 for each ~~no~~ file that stores 2 values, ~~No~~ Node i.name = i ∀ i ∈ {1 .. N}
Node i.times = access [i])

→ Queu Q1 = all nodes queued in the order of the list i.e. the elements are sorted by access frequeny top - lowest frequeny

→ Queu Q2 = empty queu

→ While (q1 or q2 is not empty):

- Check the front of ~~both~~ q1 and q2 and set node A to the one with the lower .times value and deque node A
- Check the front of q1 and q2 again and set node B to the one with the lower .times value and dequeu node B
- Create a new internal node C with A and B as its children and C.times = A.times + B.times
- Enqueu C to the rear of Q2

→ The last remaining node is the root, return the generated tree

Time complexity: Each element i.e. a node in q1 is only considered once, Moreover, since q2 consists of 'combined' nodes the worst case is when every node is merged into the same node. Since both of these O(n) ⇒ total time = O(n)

Correctness: Since this algorithm and the problem is identical to Huffman Encoding ⇒ this algo is ~~both~~ the most optimal binary tree

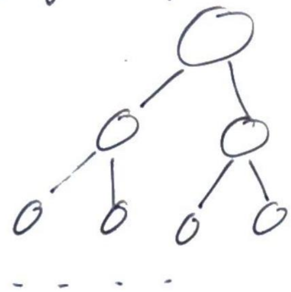
As the files can be thought of as characters, and ~~access~~ access frequency can be thought of as the frequency of appearance, and the ~~bitvector~~ length of the bit vector for each character = depth.

1 Problem 1 20 / 20

- ✓ **- 0 pts** Correct
- **- 3 pts** did not update queue
- **- 7 pts** Huffman coding not correct
- **- 10 pts** algorithm runtime not correct
- **- 15 pts** wrong answer but showed efforts
- **- 20 pts** no answer

2 Make Heap (List of n items called input):

→ Let H be an empty heap with n-elements : // an empty complete binary tree w n elements



→ Fill in the 'heap' H with values from ~~t~~ input sequentially i.e. the root = input [0], root's left child = root [1] ···
s.t. the first node on the i^{th} level (assuming the root is the 0^{th} level) is filled in by the input $[2^i - 1]$ and all ~~ot~~ nodes to its right (on the same level) are filled in sequentially

→ Let j be the last level of ~~the~~ the heap H

→ For each level from j through 0:

- For each node on that level:
  - Let T be the heap rooted at this node
  - swap this node with its children until T is a valid heap

→ return H

Time Complexity: ~~For every element in the heap, there is so~~ For elements on the bottommost level the subheap T is trivially valid ∴ 0 swaps, for the level above it ~~there~~ it takes at most 1 swap more i.e. $1+0=1$ but this can be said for all the levels above inductively ∴ in the worst case the heap ~~is has~~ $n/2^i$ nodes on the i^{th} level (assuming the i^{th} level is the i^{th} level from the bottom). ∴ Total runtime $= \sum_{i}^{\lfloor \log n} \frac{n}{2^i}(i-1) \le n \sum_{i=1}^{\infty} \frac{1}{2^i}$

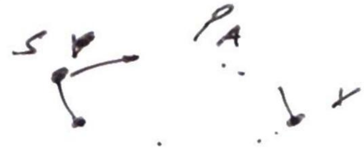
⊗ but $\sum_{i=1}^{\infty} 1/2^i \to 0$ ∴ $O(n)$. Correctness is obvious as ~~at~~ each level T is correct and the final T is our heap H and hence is valid.

2 Problem 2 **20 / 20**

- ✓ **- 0 pts** Correct
- **- 4 pts** did not update the index
- **- 10 pts** algorithm runtime not correct
- **- 15 pts** wrong answer but showed efforts
- **- 20 pts** no answer

3. a) Let P_i be the shortest path from vertex s to a vertex $i \in V(G)$ and $i \neq s$ (can be found using Djikstra's)
P_i must exist $\forall\ i \in V$ and $i \neq s$ as s is said to have a path to every other vertex in G.
Let T be the union of all P_i i.e. $T = \bigcup_{i \in V} P_i$.

· T is a subgraph of G that contains all of G's vertices
Now we will prove by contradiction that T is a tree.
Assume T is not a tree i.e. T has a cycle.
This implies for some vertex $v \in G$, there exist 2 paths from s to v



But since only 1 of these P_B could have been chosen as our shortest path, both P_A and P_B cannot be in T $\therefore$ T cannot have a cycle. Therefore T is a directed tree in G w root s s.t. the path from s to any node in T is the shortest path from s to said node in G. QED.

b) ~~Let P_K: $G_K = G_{K+1}$ where~~ ~~Let~~ G_i $\forall\ i \in \mathbb{N}$ is the graph G with edges' weights squared i times
Let T_i be the subgraph of G_i that satisfies the properties mentioned in a)
$\therefore$ Let P_K: $T_K = T_{K+1}$ (Assume this - question says so)
$\Rightarrow$ for every path the weightiest edge has a weight more than the sum of all edges in the shortest path from u to v.
$\Rightarrow$ $\underset{\text{edge in every other path}}{\text{weightiest}} \geq$ sum of all edges in the shortest path
square both sides
$(e_{weightiest})^2 \geq \Big(\sum_{i \in \text{shortest path from } u \text{ to } v} e_i\Big)^2 \geq \sum e_i^2$, but this implies
T_{K+1} T_{K+2}

3.1 3.a **5 / 5**

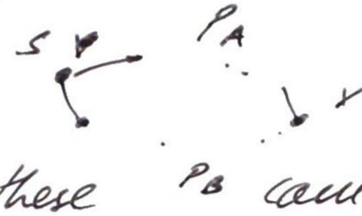
✓ **- 0 pts** Correct

- 2 pts Need more explanation

- 5 pts No answer found

3. a) Let P_i be the shortest path from vertex s to a vertex $i \in V(G)$ and $i \neq s$ (can be found using Djikstra's)
P_i must exist $\forall\ i \in V$ and $i \neq s$ as s is said to have a path to every other vertex in G.
Let T be the union of all P_i i.e. $T = \bigcup_{i \in V} P_i$.

· T is a subgraph of G that contains all of G's vertices
Now we will prove by contradiction that T is a tree.
Assume T is not a tree i.e. T has a cycle.
This implies for some vertex $v \in G$, there exist 2 paths from s to v



But since only 1 of these could have been chosen as our shortest path, both P_A and P_B cannot be in T $\therefore$ T cannot have a cycle. Therefore T is a directed tree in G w/ root s s.t. the path from s to any node in T is the shortest path from s to said node in G. QED.

b) ~~Let P_K: $G_K = G_{K+1}$ where~~ G_i $\forall\ i \in \mathbb{N}$ is the graph G with edges' weights squared ~~k~~ i times
Let T_i be the subgraph of G_i that satisfies the properties mentioned in a)
$\therefore$ Let P_K: $T_K = T_{K+1}$ (Assume this - question says so)
$\Rightarrow$ for every path the weightiest edge has a weight more than the sum of all edges in the shortest path from u to v.
$\Rightarrow$ weightiest edge in every other path $\geq$ sum of all edges in the shortest path
square both sides
$$(e_{weightiest})^2 \geq \Big(\sum_{i \in \text{shortest path from } u \text{ to } v} e_i\Big)^2 \geq \sum e_i^2$$, but this implies
T_{K+1} T_{K+2}

3.2 3.b 10 / 10

- ✓ **- 0 pts** Correct
- **- 5 pts** Click here to replace this description.
- **- 3 pts** Need more explanation
- **- 10 pts** No answer found

$\therefore P_{K+1}$ is true

Hence $P_K \Rightarrow P_{K+1}$ QED

c) Using the proof in b) it is easy to see that the edges chosen in the stable tree T_K' are the lightest edges as the length of every path is categorized in T_K' by the weight of it's heaviest edge and thus a tree like T_K' that guarantees the ~~shortest~~ paths from root to every vertex are the shortest paths in $G_K' \Rightarrow$ it guarantees a minimal spanning tree (as T_K' must also contain all vertices in G_K') ~~and~~ of G'

d) By the same reasoning in b) and c) the spanning trees can be ~~comp~~ compared by comparing individual edges in a G_K'-like graph to construct a union of shortest paths T_K' that is the minimal spanning tree.

4. a) ~~Def~~ Query (a, b):

return Find (a) == find (b)

Find (a):

Traverse upwards until the root is reached

return root

Union (a, b):

v = Find (a)

w = Find (b)

if v == w do nothing and return

else add the vertex from v and w that is a root to a tree with lower height as a child of the other

3.3 3.c 7 / 10

- **- 0 pts** Correct
- ✓ **- 3 pts Need more explanation**
- **- 5 pts** Need more explanation
- **- 10 pts** No answer found

$\therefore P_{K+1}$ is true
Hence $P_K \Rightarrow P_{K+1}$ QED

c) Using the proof in b) it is easy to see that the edges chosen in the stable tree T_K' are the lightest edges as the length of every path is categorized in T_K' by the weight of it's heaviest edge and thus a tree like T_K' that guarantees the ~~shortest~~ paths from root to every vertex are the shortest paths in $G_K' \Rightarrow$ it guarantees a minimal spanning tree (as T_K' must also contain all vertices in G_K') ~~and~~ of G'

d) By the same reasoning in b) and c) the spanning trees can be ~~comp~~ compared by comparing individual edges in a G_K'-like graph to construct a union of shortest paths T_K' that is the minimal spanning tree.

4. a) ~~Def~~ Query (a, b):
return Find (a) == find (b)

Find (a):
Traverse upwards until the root is reached
return root

Union (a, b):
v = Find (a)
w = Find (b)
if v == w do nothing and return
else add the vertex from v and w that is a root to a tree with lower height as a child of the other

3.4 3.d **7 / 10**

- **- 0 pts** Correct
- ✓ **- 3 pts Need more explanation**
- **- 5 pts** Need more explanation
- **- 10 pts** No answer found

$\therefore P_{K+1}$ is true
Hence $P_K \Rightarrow P_{K+1}$ QED

c) Using the proof in b) it is easy to see that the edges chosen in the stable tree T_K' are the lightest edges as the length of every path is categorized in T_K' by the weight of it's heaviest edge and thus a tree like T_K' that guarantees the ~~shortest~~ paths from root to every vertex are the shortest paths in $G_K' \Rightarrow$ it guarantees a minimal spanning tree (as T_K' must also contain all vertices in G_K') ~~and~~ of G'

d) By the same reasoning in b) and c) the spanning trees can be ~~comp~~ compared by comparing individual edges in a G_K'-like graph to construct a union of shortest paths T_K' that is the minimal spanning tree.

4. a) ~~Def~~ Query (a, b):
   return Find (a) == find (b)

Find (a):
  Traverse upwards until the root is reached
  return root

Union (a, b):
  v = Find (a)
  w = Find (b)
  if v == w do nothing and return
  else add the vertex from v and w that is a root to a tree with lower height as a child of the other

The processing algorithm can call the Union(a, b) and ~~Find(a,b)~~ Query(a, b) functions to process all commands.

Time complexity: Since the time complexity of find is the bottleneck in the algorithm and is of $O(h)$ of the tree that it traverses, it suffices to understand that union ensures $h \leq \log n$ always. $\therefore$ Total complexity $= O(n \log n)$
Correctness is trivial from the understanding that if find returns the same answer 2 elements are part of the same tree i.e. the same set and union ensures that when 2 trees are merged they make one new roughly 'balanced' tree.

b) Let list l be an ordering of the edges in G in ascending order of weights
Let T be an empty tree
For every element (u, v) in l where this represents an edge ~~from~~ between u and v:
    if(Query(a, b))
        do nothing
    Else Union(a, b)
Return T

Time complexity: There are |E| commands (Query and/or Union) and each command takes log|V| $\therefore$
$O(|E| \log |V|)$.
Correctness: follows from the connectedness of G and the way union works (in set ops)

4.1 4.a **15 / 15**

- ✓ **- 0 pts** Correct
- **- 15 pts** No answer found
- **- 5 pts** Lack step-to-step description of algorithm
- **- 10 pts** wrong answer but showed efforts

The processing algorithm can call the Union(a, b) and ~~Find(a, b)~~ Query(a, b) functions to process all commands.

Time complexity: Since the time complexity of find is the bottleneck in the algorithm and is of O(h) of the tree that it traverses, it suffices to understand that union ensures $h \leq \log n$ always. $\therefore$ Total complexity = $O(n \log n)$

Correctness is trivial from the understanding that if find returns the same answer 2 elements are part of the same tree i.e. the same set and union ensures that when 2 trees are ~~s~~ merged they make one new roughly 'balanced' tree.

b) Let list l be an ordering of the edges in G in ascending order of weights
Let T be an empty tree
For every element (u, v) in l where this represents an edge ~~from~~ between u and ~~a~~ v:
    if (Query (a, b))
        do nothing
    Else Union (a, b)
Return T

Time complexity: There are |E| commands (Query and/or Union) and each command takes log|V| $\therefore$
$O(|E| \log |V|)$.

Correctness: follows from the connectedness of G and the way union works (in set ops)

4.2 4.b 10 / 10

✓ **- 0 pts** **Correct**

- 3 pts Minor mistake

- 5 pts Lack step-to-step description of algorithm

- 10 pts No answer found