CS170 — Fall 2017— Homework 5 Solutions

Jonathan Sun, SID 25020651

0. Who Did You Work With?

Collaborators: Kevin Vo, Aleem Zaki, Jeremy Ou

1. Scheduling Homeworks

(a) It is incorrect because a counterexample is one homework assignment due in 1 hour (worth one point) and two homework assignments due in 2 hours (worth two points each). If we consider the one whose deadline is the earliest and schedule it at the earliest available time, then we will only get the assignment due in 1 hour done and one of the assignments due in 2 hours done. This gives us 3 points. However, we could have gotten 4 points if we did both assignments due in 2 hours.

(b) It is incorrect because a counterexample is one homework assignment due in 1 hour (worth one point) and one homework assignments due in 2 hours (worth two points). If we consider the one whose points is the highest and schedule it at the earliest available time, then we will only get the assignment due in 2 hours done. This gives us 2 points. However, we could have gotten 3 points if we did the assignment due in 1 hour first before doing the assignment due in 2 hours.

(c) This greedy algorithm is correct because we want to minimize the number of "gaps" of time spent not doing homework. When using this algorithm, the student will still be able to do the homework assignments worth the most points before their respected deadlines at each time frame. This will also solve the problem in part (b) where there can be "gaps" of time spent wasted because you will still be packing in as many working hours as possible.

2. Graph Coloring

(a) To show that this algorithm terminates in a finite number of steps, I will use an example. Imagine the worst case, which is where there is a red vertex with 85 red vertex neighbors and 84 blue vertex neighbors. This red vertex is considered a bad vertex because it has at least 85 neighbors of its own color. After reversing the color, there are still 85 red vertex neighbors and 84 blue vertex neighbors. However, this vertex is a good vertex because it has less than 85 neighbors of its own color. Therefore, this example shows that even in the worst case scenario, switching the color of a bad vertex will remove only 1 bad vertex from B. In other words, you can remove n bad vertices from B where n is the difference between the number of neighbors of its own color and number of neighbors of the the color. Therefore, each iteration of the loop leads to the decrease and so the algorithm will terminate when it reaches 0.

(b) The algorithm will terminate after at most $|E|$ iterations of the loop because in each iteration of the loop, it will remove at least 1 bad vertex, which also means 1 bad edge. Therefore, the best case will have all the bad vertices removed in just one iteration. The worst case will have only 1 bad vertex removed, which means 1 edge at a time. This worst case will need all edges removed with 1 removed in each iteration, meaning that there will be $|E|$ iterations needed.

3. 170-Graph

Main Idea:
The main idea is that we use a modified version of DFS because DFS will end up checking all vertices as well as keeping track of the previsit and postvisit numbers. The previsit and postvisit numbers are important because we want to find the subgraph for $|V'|$ to be as large as possible. Therefore, if there are multiple subgraphs, then the subgraphs with the largest difference between the previsit and postvisit numbers for the "source" node will be the largest subgraph. For every node that we are on in the DFS, we will keep a counter to check if that node has at least 170 neighbors. If so, then we keep track of that node, in a data structure that keeps track of which nodes are in our current subgraph. In the end, if we get multiple subgraphs, then I can just find the largest subgraph easily.

Run Time:
$O(|V||E| + |E|^2)$. This is because we are just doing DFS, which has $O(|V| + |E|)$ runtime and checking if a node has at least 170 neighbors, which is $O(|E|)$ time. Therefore, the runtime will be $O((|V| + |E|)|E|) = O(|V||E| + |E|^2)$.

4. Weighted Set Cover

Proof of Correctness:
To prove that if there is a solution with cost k then the above algorithm will find a solution with cost $O(k \log n)$, I let k be the cost of optimal solution. From the idea above, at each iteration we have covered $i-1$ elements before it, which means that the cost of the ith element is at most worth $k \times \frac{1}{n-(i-1)}$. Since the S_i that is picked in each iteration has the largest ratio, the cost of the ith element is at most $k \times \frac{1}{n-(i-1)}$ and so the cost can be represented as $\sum_{i=1}^{n} \frac{k}{n-i+1} = k \times \left(\frac{1}{n} + ... + \frac{1}{2} + 1\right)$ and this is about $O(k \log n)$ runtime. □

5. Quaternary Huffman

Main Idea:

The main idea is to modify Huffman's algorithm so that instead of working with a binary tree system, you work with a quarternary tree system. This would mean that a node can have up to 4 children instead of the normal 2 children. Therefore, the strategy to create the quarternary tree system will be the same strategy for the most part as the binary tree system for normal Huffman's algorithm. However, after creating the quartnary tree, we may need to rebalance the tree since we want to maximize the number of letters that can be represented with minimal bits.

Proof of Correctness:

To prove that the algorithm will achieve the maximum possible compression, I will use the proof for normal Huffman's algorithm provided in lecture (lemma that the two nodes with the lowest frequencies are at the greatest depth and lemma that nodes with the lowest frequencies are siblings in an optimal tree) to justify that ordinary Huffman's algorithm will achieve the maximum possible compression. So, I will just need to prove that my main idea will also achieve maximum possible compression. My algorithm idea is to work with a quarternary tree, which can represent up to 4 values in one layer. So, my algorithm will maximize the use of this data structure by putting the least frequently used characters at the lowest layer of the quarternary tree and will prioritize character spots in upper layers for characters that are more frequently used. This alone will allow me to assign the most frequently used characters with shorter codewords than those used less frequently. Once this part is done, it is important for me to rebalance the tree since there may be unused branches above the characters in the lower layers. So, rebalancing the tree would mean that if a layer has not maximized its branches, then the algorithm will bubble up the most used characters that is below that layer. This way, the characters that get bubbled up will get to be able to reduce their encodings. □

6. Simple and Naive Cluster Analysis

Four-Part Solution:

Main Idea:

The main idea is to have two sets and to focus on the dissimilar messages although it can also be done with the similar messages. As we go though the messages, we check if the message that we are on has any dissimilarities with any other messages in either sets. Then we will put that message into the set with the fewest dissimilarities. If there are no dissimilarities in either sets for that message, then we can arbitrarily put it into either set.

Pseudocode:

```
clusterAnalysis(List of messages):
  set1 = set()
  set2 = set()
  for each message:
    if message has fewer dissimilarities in set1 than set2:
      set1.add(message)
    if message has more dissimilarities in set1 than set2:
      set2.add(message)
    else:
      add message to either sets
  return set1 and set2
```

Proof of Correctness:

The pseudocode works because it is just a simple algorithm that checks if a message has fewer similarities in one set over the other and assigning it to the set that has fewer similarities. This will end up checking if there are any dissimilarities of the node that we are on with those that came before it. This essentially means that we will split the number of dissimilar edges in half because we are separating them. This will separate the pairs of dissimilar messages quickly in a simple way. □

Run Time:

$O(n^2)$.

Justification:

The runtime is $O(n^2)$ because in each iteration of the algorithm, the number of checks needed is proportional to the iteration number. This means that the number of checks being done is equivalent to $1 + 2 + 3 + ... + n$. This can be represented by the arithmetic sum, $\sum_{n=1}^{n} n = \frac{1}{2}n(n+1)$. This converges to $O(n^2)$ runtime.