Jason Less
404-640-158
CS180

CS180 Homework 4

Question 1

My initial thought was to construct the array as a max heap, so that the files with the larger access frequency counts would be closer to the root of the tree; and thus, resulting in the minimum total cost of all accesses. However, the problem asks that the files only be placed at the leaves and not at any root of any subtrees.

I started off with some base cases: (1) single file [1], (2) two files [1, 2], and (3) three files [1, 2, 3]. For the first case, there is a single file, so just place it at the root of the binary tree (i.e. leave), done. For the second case, as there are two files, one of these files cannot be placed at the root of the tree. Thus, place down a dummy node, and then place 2 (i.e. the most frequently accessed file) at the right child node. At this point, there is just one node left to place, so place 1 in the left child node, done. For the last case, there are three files, so place dummy node at the root, and place 3 at the right child node. Now there are two files left, so can't place any in the left child node, so place a dummy node there, and place 2 in the new dummy node's right subtree. From there, there is just one file left, place it in the new dummy node's left subtree, done.

From these base cases, the idea of placing the nodes is quite simple: (1) if there is one file, place it at the root of the current subtree, (2) if there are two files, create a dummy node, and place the files at the two children of the dummy node, and (3) if there are three or more files, create a dummy node, place the most frequently accessed file in the right child node, and then repeat (2) for the dummy node's left subtree.

As the array is sorted in increasing order, this would just require one pass through the array starting at the back (where the back of the array signifies the nodes that should appear near the root of the tree), which would take O(n) time as required.

Algorithm:

1. If array is empty, return nullptr (i.e. empty binary tree), done
2. If array is of size 1, return tree with the single value as the root, done
3. Initialize root with value of -1
4. Create curr ptr to pt to the root
5. for (i = last elem in arr through i = 0)
    a. if (i >= 2)
        i. curr->right = new node with value of arr[i]
        ii. curr->left = new node with value of -1
        iii. curr = curr->left
    b. else
        i. curr->right = new node with value of arr[i]
        ii. curr->left = new node with value of arr[i – 1]
        iii. break
6. Return root, done

Jason Less
404-640-158
CS180

I know it is not required, but I was interested in writing the code for this problem, so below is a snippet of C++ code of my implementation:

```cpp
#include <iostream>
#include <vector>
using namespace std;

struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode (int val) : val(val), left(nullptr), right(nullptr) {}
};

TreeNode* buildMinTree(vector<int>& arr) {
    if (arr.empty()) return nullptr;
    if (arr.size() == 1) return new TreeNode(arr[0]);

    TreeNode* root = new TreeNode(-1);
    TreeNode* curr = root;
    for (int i = (int)arr.size() - 1; i >= 0; i--) {
        if (i >= 2) {
            curr->right = new TreeNode(arr[i]);
            curr->left = new TreeNode(-1);
            curr = curr->left;
        }
        else {
            curr->right = new TreeNode(arr[i]);
            curr->left = new TreeNode(arr[i - 1]);
            break;
        }
    }

    return root;
}
```

Jason Less
404-640-158
CS180

Question 2

The idea behind constructing such a heap is to think of the array as a complete binary tree, and to perform shuffling of elements in the array in-place such that it resembles a heap. Starting at the bottom-most right-most subtree (i.e. the last element in the array), we want to convert it to a max heap, and then move on to the next bottom-most right-most subtree, and convert it to a max heap, and so on.

At each subtree, we want to convert the subtree into a max heap, which can be carried out by the heapification process, which compares the root node's value to the value's of each of its children, and then swaps the largest value of the three nodes to the root. If this results in a new root node, then the node that was swapped to one of the children node needs to sift down the subtree until the subtree is a max heap again.

Note that we don't need to do start at the bottom-most right-most node of the tree, as these are leaf nodes (and thus are already max heaps). Thus, we want to instead start at the bottom-most right most-node that has at least one child, which is at index = size/2 – 1. Thus, the algorithm to convert the array into a max heap looks like the below:

ConstructHeap Algorithm

1. for (i = size/2 - 1; i >= 0; i--) heapify(arr, i)

Heapify Algorithm

1. if arr[left] > arr[index], then largestIndex = left
2. if arr[right] > arr[index], then largestIndex = right
3. if largestIndex changed from the previous two steps:
    a. Swap arr[index] with arr[largestIndex]
    b. heapify(arr, largestIndex)

Complexity Analysis

Initially, it seems that the process of building a heap should be O(N log N), as we loop through N elements in the list, and each heapify step looks to be O(log N) time complexity. This is because the node that sifts down the tree for the current heapify at worst would move down to the leaf (i.e. log N levels down in the tree). However, the algorithm actually takes O(N) time, as heapify is not O(log N) for all the nodes. The running time depends on how far an element might have to sift down (i.e. the height of the element in the tree). At the bottom-most level, there are 2^h nodes, but all the nodes on this level are leaves, and thus we don't call heapify for this level (i.e. zero work done). The next level, consists of 2^{h-1} nodes, and each might have to sift down 1 level. The 3rd level from the bottom consists of 2^{h-2} nodes, and each might have to sift down 2 levels. The following would lead to a long mathematical proof, but as it can be seen, not all the heapify operations are O(log N); and therefore, the time complexity is O(N).

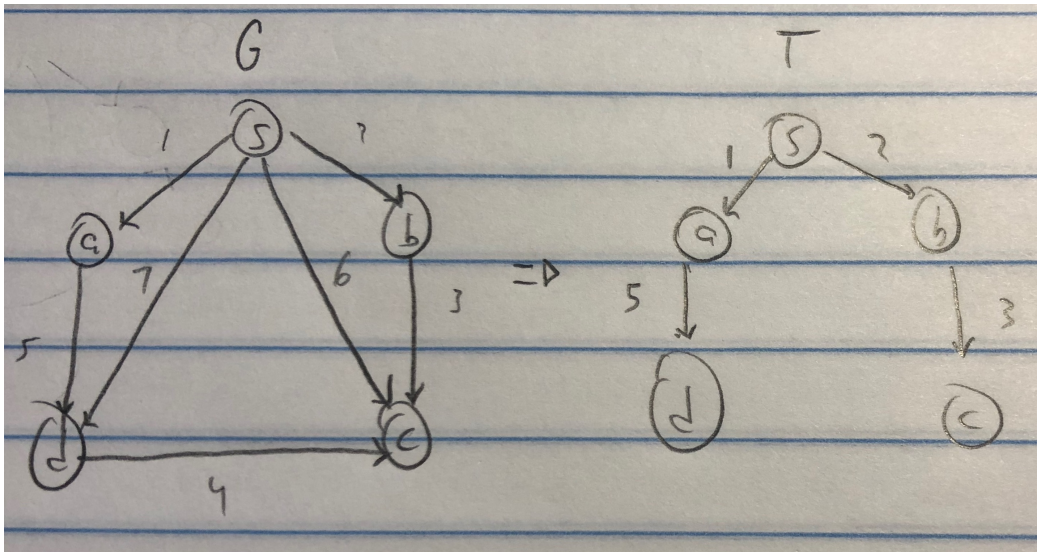
Jason Less
404-640-158
CS180

Question 3

(a)

The existence of a directed tree T in G with root s as a sub graph in G can be proved by applying Dijkstra's algorithm. Dijkstra's algorithm states that given a source node s, it will find the shortest paths from the source to all other nodes in the graph, producing a shortest-path tree with node s as the root of the tree. Therefore, it can be stated that there exists a directed tree T in G with root s as a subgraph of G with the property that the path from s to any other node v in T is the shortest path from s to v in G.

The example below shows a directed graph with node s being able to reach all other vertices of the graph. After applying Dijkstra's algorithm, it generates a shortest-path tree where the path from s to all other nodes is the shortest path from s to the other nodes in the graph G:



(b)

Given an initial shortest path tree, squaring the weights of the edges for the first time could change the shortest path tree (as stated in the problem description) depending on the path that has the largest weight. This is because the largest weight dominates the entire path. Further squaring of the weights of the edges will not change the shortest path tree any further for the same reason. The path that has the largest weight will continue to grow at a rate far faster than any of the other weights. Therefore, the problem really boils down to the path that has the largest weight (i.e. comparison rather than addition). Using the same example, path 1 (with edges having weights 5 and 5) has value 10, and path 2 (with edges having weights 7 and 2) has value 9. Thus, path 2 is the shortest path. After squaring the weights once, path 1 has value 50 and path 2 has value 52. Thus, path 1 is now the shortest path. This will remain the shortest path after further squaring of weights. To continue, after squaring twice, path 1 has value 1250 and path 2 has value 2417. Once more, path 1 has value 781,250 and path 2 has value 5,765,057.

Jason Less
404-640-158
CS180

(c)

This can be shown using proof by contradiction via Prim's algorithm. Let T_k' be the minimum spanning tree of graph G. Also assume that there exists a spanning tree T_k, which needs to be proved that it is the minimum spanning tree of graph G. If $T_k' = T_k$, then T_k is in fact the minimum spanning tree of graph G. So the contradiction is that $T_k' \neq T_k$. Assume that on the kth iteration of Prim's algorithm, there is an edge E = (u, v) chosen that is not in T_k'. Since, T_k' is the MST connecting all the nodes, there should exist a path P from u to v in G. Also assume there exists an edge E' on path P such that one endpoint is in the tree generated at the (k-1)th iteration of the algorithm, and the other is not. If w(E') < w(E), then Prim's algorithm would have chosen E' on the kth iteration of the algorithm; thus, w(E') >= w(E). This is a contradiction as T_k' is the MST containing E', which should have a smaller weight than E. Hence, T_k' must be the MST in G.