CS180 HW4

Xilai Zhang

TOTAL POINTS

100 / 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 10 / 10

✓ **- 0 pts Correct**

- 3 pts Need more explanation

- 5 pts Need more explanation

- 10 pts No answer found

3.4 3.d 10 / 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

1.

Initialize an empty array B of nodes, each has a value.
Initialize a adjacency list D that stores edge information.
While f is not empty and B has more than 1 node:

If B is empty:

Erase the two smallest values in f, put their sum in a node C and append the node C at the end of array B. Put edges (outgoing from C) between C and the two values in D.

Else if B only has one item:

Find the two smallest value among the three(first two elements of f and first element of B), erase the two smallest values and put their sum in a struct C, append the struct C at the end of array B. Put edges (outgoing from C) between C and the two elements in D.

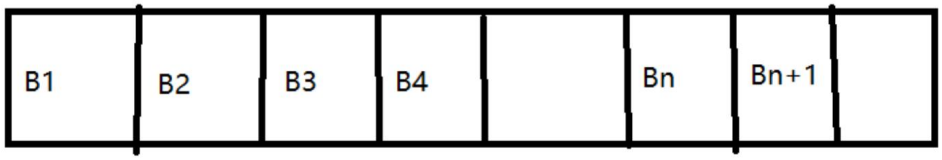
Else:

Find the two smallest value among the four (first two elements of f and first two element of B), erase the two smallest values and put their sum in a struct C, append the struct C at the end of array B. Put edges (outgoing from C) between C and the two elements in D.

Return D.

Correctness:

The correctness of the algorithm depends on the fact that the second array increase monotonically. To prove that this claim is true, we first assume it is wrong and then try to find contradiction. Let us assume A1, A2, A3, A4 are elements in array x, and assume B1,B2,B3,B4 are elements in array y.



Now suppose array y does not increase monotonically, and without loss of generality, we assume Bn+1 is the first element in array y that is not increasing. That is, Bn+1 is at least smaller than Bn, namely, Bn+1<Bn. Now, we have the following cases:

(1) Bn+1 is generated by taking the first element from array x and the first element from array y

Without loss of generality, let us assume Bn+1 is formed by A3 and B3, that is, A3+B3=Bn+1. Now, Bn must have been generated in one of the following two ways:

(i) Bn is also generated by taking the first element from array x and the first

element from array y, and in our example, Bn=A2+B2. Since both array are sorted, A3>A2 and B3>B2, thus Bn+1=A3+B3> A2+B2=Bn. This contradicts with our assumption that Bn+1<Bn

(ii)Bn is generated by taking two elements from one array. Without loss of generality, let us assume Bn is formed by taking two elements from array x. In this case, Bn=A1+A2. Since Bn choose A1 and A2 when B3 is present, we know A1<A2<B3 and A1<A2<A3. Thus Bn=A1+A2<B3+A3<Bn+1. Again this contradicts with our assumption that Bn+1<Bn.

(2) Bn+1 is generated by taking two elements from one array. Without loss of generality, let us assume Bn+1 is formed by taking two elements from array x. In this case, Bn+1=A3+A4. Now, Bn must have been generated in one of the following two ways:

(i) Bn is generated by taking the first element from array x and the first element from array y, and in our example, Bn=A2+B2. Since Bn used B2 and A2 when A3 is present, we know B2<A3<A4 and A2<A3<A4, thus Bn=A2+B2 < A3+A4=Bn+1. This contradicts with our assumption that Bn+1<Bn

(ii)Bn is generated by taking two elements from one array. Without loss of generality, let us assume Bn is formed by taking two elements from array y. In this case, Bn=B1+B2. Since Bn choose B1 and B2 when A3 is present, we know B1<B2<A3<A4. Thus Bn=B1+B2<A3+A4<Bn+1. Again this contradicts with our assumption that Bn+1<Bn.

Thus we have proven that Bn+1<Bn will always lead to a contradiction. Thus Bn+1 must be greater than Bn for all n. And thus array y will be monotonically increasing.

Complexity:

The crux of this algorithm takes the two smallest frequencies, combine them, and put them at the bottom of the binary tree. In the worst case each iteration we compare among the four elements (first two elements of f and first two elements of B), but it is still constant time. The number of iterations we need is n-1 because each time we combine we reduce the total number of nodes by 1. In the end we are left with only one node and we are done. So in total the time complexity is constant * O(N)=O(N).

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.

Put the n items in an array A.
Initialize i to be n/2-1. //we can do this because the bottom layer is good
Initialize j to store temp value.
While i is not 0:
    j=i.
    while true:
        Compare A[i] with its two children A[2i] and A[2i+1]:
            If A[i] is bigger than both of its children:
                break
            If A[i] is smaller than any of its children:
                Swap A[i] with the bigger of A[2i] and A[2i+1].
                i= the index of the bigger children. (either 2i or 2i+1)
                if i is greater than n/2-1:
                    break
    i=j
    i=i-1
return A.

Correctness:

The algorithm does not consider the bottom layer. Starting from the second from bottom layer, each node, if out of order, will be swapped with the bigger of its children. For each node, it triggers a series of top to bottom swaps until we reach the bottom of the heap or the node is bigger than both of its children. Because the bigger of its children is swapped up, it is guaranteed that the bigger children will be bigger than the other children and all descendents of the other children.

Complexity:

The swap with the bigger of its children will take constant time. The maximum number of operations to perform for each node depends on the depth from that node to the bottom. If we multiply the number of elements on each layer and that layer's distance to the bottom, we will get the total complexity as

$$\frac{n}{4} * 1 + \frac{n}{8} * 2 + \frac{n}{16} * 3 + \cdots + \frac{n}{2^{h+1}} * h$$

Where h is the depth from that layer to the bottom layer. Now we know this series is monotonically increasing and thus it is smaller than the series if we take h to infinity. Thus the number will be bounded by the value if we take h to be infinity. To calculate the value when h is infinity, we can employ taylor series, namely, geometric series:

$\frac{1}{1-x} = \sum_{n=0}^{n=\infty} x^n$ and its derivative $\frac{1}{(1-x)^2} = \sum_{n=1}^{n=\infty} nx^{n-1}$

Thus $n * \sum_{m=0}^{m=\infty} (m+1)(\frac{1}{2})^{m+2} = n * \frac{1}{4} * \sum_{m=1}^{m=\infty} m(\frac{1}{2})^{m-1} = n * \frac{1}{4} * \frac{1}{(1-\frac{1}{2})^2} = n$

Thus the total complexity will be bounded by constant * O(N) which is O(N).

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.

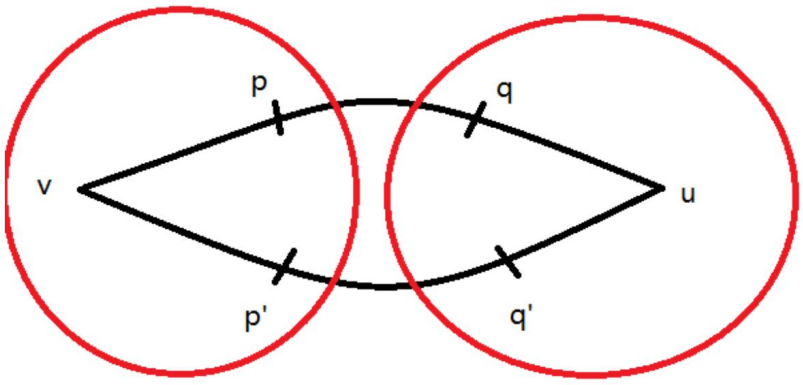
(a) We can perform dijkstra's algorithm starting from s and reache every node, this way it is guaranteed that all paths from s are shortest path. We prove this generates a tree by claiming that it is connected and acyclic. First it is connected because dijkstra's algorithm travels to all the nodes. Secondly dijkstra's algorithm ensures that a visited node will not be visited again. Once a node is visited, it is removed from further consideration of setting up edge, and thus there is no way to set up the last edge that completes the cycle. Thus the graph we generated by dijkstra's algorithm is connected and acyclic, and is thus a tree.

(b) We can prove the minimum path converges to the bottleneck path(the path with the longest edge being the smallest). Say the bottleneck path has edges x1,x2,…xn with x1 being the longest, and assume the other path has edges y1,y2,…yn with y1 being the longest edge. We know that y1 is the longest edge among them. Now assume we square the edges enough time and compare ${x_1}^{2k} + {x_2}^{2k} + \cdots + {x_n}^{2k}$ *and* ${y_1}^{2k} + {y_2}^{2k} + \cdots + {y_n}^{2k}$ when k goes to infinity.

$$\lim_{k\to\infty} \frac{{x_1}^{2k} + {x_2}^{2k} + \cdots + {x_n}^{2k}}{{y_1}^{2k} + {y_2}^{2k} + \cdots + {y_n}^{2k}} = \lim_{k\to\infty} \frac{\frac{{x_1}^{2k}}{{y_1}^{2k}} + \frac{{x_2}^{2k}}{{y_1}^{2k}} + \cdots + \frac{{x_n}^{2k}}{{y_1}^{2k}}}{\frac{{y_1}^{2k}}{{y_1}^{2k}} + \frac{{y_2}^{2k}}{{y_1}^{2k}} + \cdots + \frac{{y_n}^{2k}}{{y_1}^{2k}}} = \lim_{k\to\infty} \frac{0}{1} = 0 < 1$$

Thus as we keep squaring the weight of edges, the longest edge in each path will dominate and bottleneck path will be smaller than all other paths.

(c) We know that for tree T_k' all the paths from v to other nodes are bottleneck paths. Now we claim that the minimum spanning tree cannot have any path from v to other nodes that is not bottleneck path. Suppose there is a path in minimum spanning tree T from node v to node u such that it is not a bottleneck path. Assume the longest edge on T from v to u is edge pq. Assume the longest edge on T_k' is p'q'. Now we know p'q' is smaller than pq.



Now consider the cut of the two red circles(nodes v, p, p' and nodes u,q,q'). According to minimum spanning tree, only the smallest edge between two cuts is included. However, the smallest edge between the two cuts is p'q', but instead pq which is greater than p'q' is included in the minimum spanning tree T. Thus this is a

3.1 3.a 5 / 5

✓ **- 0 pts** **Correct**

- 2 pts Need more explanation

- 5 pts No answer found

3.

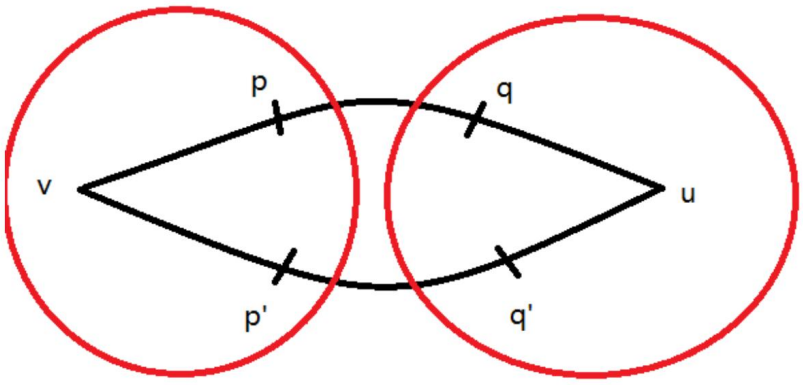
(a) We can perform dijkstra's algorithm starting from s and reache every node, this way it is guaranteed that all paths from s are shortest path. We prove this generates a tree by claiming that it is connected and acyclic. First it is connected because dijkstra's algorithm travels to all the nodes. Secondly dijkstra's algorithm ensures that a visited node will not be visited again. Once a node is visited, it is removed from further consideration of setting up edge, and thus there is no way to set up the last edge that completes the cycle. Thus the graph we generated by dijkstra's algorithm is connected and acyclic, and is thus a tree.

(b) We can prove the minimum path converges to the bottleneck path(the path with the longest edge being the smallest). Say the bottleneck path has edges x1,x2,…xn with x1 being the longest, and assume the other path has edges y1,y2,…yn with y1 being the longest edge. We know that y1 is the longest edge among them. Now assume we square the edges enough time and compare ${x_1}^{2k} + {x_2}^{2k} + \cdots + {x_n}^{2k}$ *and* ${y_1}^{2k} + {y_2}^{2k} + \cdots + {y_n}^{2k}$ when k goes to infinity.

$$\lim_{k\to\infty} \frac{{x_1}^{2k} + {x_2}^{2k} + \cdots + {x_n}^{2k}}{{y_1}^{2k} + {y_2}^{2k} + \cdots + {y_n}^{2k}} = \lim_{k\to\infty} \frac{\frac{{x_1}^{2k}}{{y_1}^{2k}} + \frac{{x_2}^{2k}}{{y_1}^{2k}} + \cdots + \frac{{x_n}^{2k}}{{y_1}^{2k}}}{\frac{{y_1}^{2k}}{{y_1}^{2k}} + \frac{{y_2}^{2k}}{{y_1}^{2k}} + \cdots + \frac{{y_n}^{2k}}{{y_1}^{2k}}} = \lim_{k\to\infty} \frac{0}{1} = 0 < 1$$

Thus as we keep squaring the weight of edges, the longest edge in each path will dominate and bottleneck path will be smaller than all other paths.

(c) We know that for tree T_k' all the paths from v to other nodes are bottleneck paths. Now we claim that the minimum spanning tree cannot have any path from v to other nodes that is not bottleneck path. Suppose there is a path in minimum spanning tree T from node v to node u such that it is not a bottleneck path. Assume the longest edge on T from v to u is edge pq. Assume the longest edge on T_k' is p'q'. Now we know p'q' is smaller than pq.



Now consider the cut of the two red circles(nodes v, p, p' and nodes u,q,q'). According to minimum spanning tree, only the smallest edge between two cuts is included. However, the smallest edge between the two cuts is p'q', but instead pq which is greater than p'q' is included in the minimum spanning tree T. Thus this is a

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.

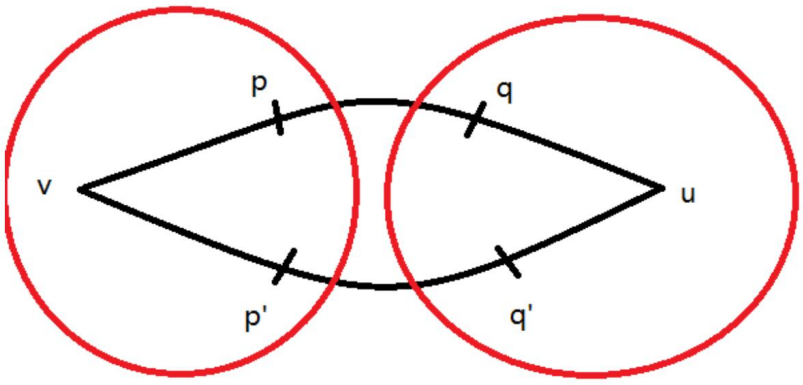
(a) We can perform dijkstra's algorithm starting from s and reache every node, this way it is guaranteed that all paths from s are shortest path. We prove this generates a tree by claiming that it is connected and acyclic. First it is connected because dijkstra's algorithm travels to all the nodes. Secondly dijkstra's algorithm ensures that a visited node will not be visited again. Once a node is visited, it is removed from further consideration of setting up edge, and thus there is no way to set up the last edge that completes the cycle. Thus the graph we generated by dijkstra's algorithm is connected and acyclic, and is thus a tree.

(b) We can prove the minimum path converges to the bottleneck path(the path with the longest edge being the smallest). Say the bottleneck path has edges x1,x2,…xn with x1 being the longest, and assume the other path has edges y1,y2,…yn with y1 being the longest edge. We know that y1 is the longest edge among them. Now assume we square the edges enough time and compare ${x_1}^{2k} + {x_2}^{2k} + \cdots + {x_n}^{2k}$ *and* ${y_1}^{2k} + {y_2}^{2k} + \cdots + {y_n}^{2k}$ when k goes to infinity.

$$\lim_{k\to\infty} \frac{{x_1}^{2k} + {x_2}^{2k} + \cdots + {x_n}^{2k}}{{y_1}^{2k} + {y_2}^{2k} + \cdots + {y_n}^{2k}} = \lim_{k\to\infty} \frac{\frac{{x_1}^{2k}}{{y_1}^{2k}} + \frac{{x_2}^{2k}}{{y_1}^{2k}} + \cdots + \frac{{x_n}^{2k}}{{y_1}^{2k}}}{\frac{{y_1}^{2k}}{{y_1}^{2k}} + \frac{{y_2}^{2k}}{{y_1}^{2k}} + \cdots + \frac{{y_n}^{2k}}{{y_1}^{2k}}} = \lim_{k\to\infty} \frac{0}{1} = 0 < 1$$

Thus as we keep squaring the weight of edges, the longest edge in each path will dominate and bottleneck path will be smaller than all other paths.

(c) We know that for tree T_k' all the paths from v to other nodes are bottleneck paths. Now we claim that the minimum spanning tree cannot have any path from v to other nodes that is not bottleneck path. Suppose there is a path in minimum spanning tree T from node v to node u such that it is not a bottleneck path. Assume the longest edge on T from v to u is edge pq. Assume the longest edge on T_k' is p'q'. Now we know p'q' is smaller than pq.



Now consider the cut of the two red circles(nodes v, p, p' and nodes u,q,q'). According to minimum spanning tree, only the smallest edge between two cuts is included. However, the smallest edge between the two cuts is p'q', but instead pq which is greater than p'q' is included in the minimum spanning tree T. Thus this is a

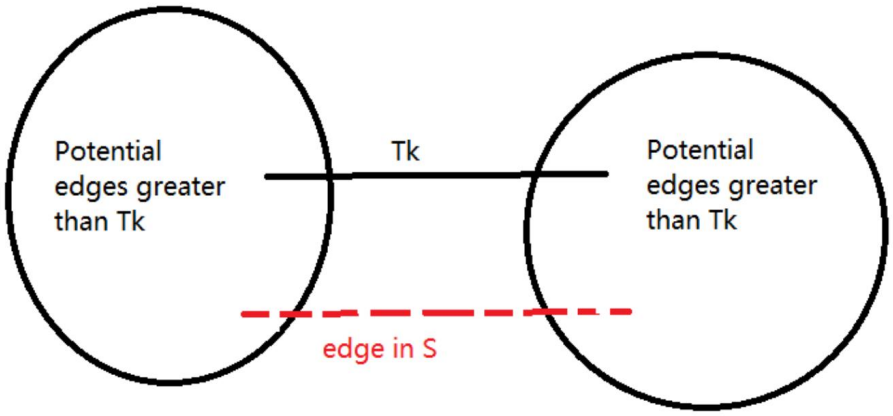
contradiction, and we see that minimum spanning tree cannot have paths from v other than bottleneck paths. Thus all the paths from v to other nodes are bottleneck paths on minimum spanning tree, and minimum spanning tree becomes the same we defined for tree T_k' .

（d）

Now we can compare two spanning trees according to the following criteria: starting from the longest edge, if the longest edges differ, then the spanning tree with the smaller longest edge is better. If the longest path is the same, we compare the second longest path. We keep comparing until the value of the edges differ or we run out of edges. That is, we find in spanning tree A which has edges sorted in decreasing order S1,S2,S3….Sn and spanning tree B which has edges sorted in decreasing order T1,T2,T3….Tn the smallest index k such that Sk is different from Tk, the tree that corresponds to the smaller of Sk and Tk is the better tree.

Now we claim that the spanning tree S that is better than all others according to the above requirement is the minimum spanning tree.

Suppose there is a minimum spanning tree T such that for index j greater than k, Sj = Tj, and for index k, Tk > Sk. According to our criteria, S is better than T. Now, we apply similar argument to part c.



According to the definition of minimum spanning tree, Tk must be the smallest and only edge between the two cuts it separates. If it is not the only edge between the cuts, there will be a cycle and violates MST definition. Thus we know all edges greater than Tk must be in the two cycles. Now consider tree S, all the edges in S that are greater than Tk are the same as all the edges in T that are greater than Tk. But we already know these edges are in the circles and do not connect the two cuts. Thus, the edge in S that connects the two cuts must be smaller than Tk. Now, this generates a contradiction because Tk is no longer the smallest edge between the two cuts. This violates the MST definition and thus this T could not exist. Thus we have proven that T cannot be a minimum spanning tree if T is better than S. And thus the tree S that is better than all other trees is the minimum spanning tree.

3.3 3.c 10 / 10

✓ **- 0 pts** **Correct**

- 3 pts Need more explanation

- 5 pts Need more explanation

- 10 pts No answer found

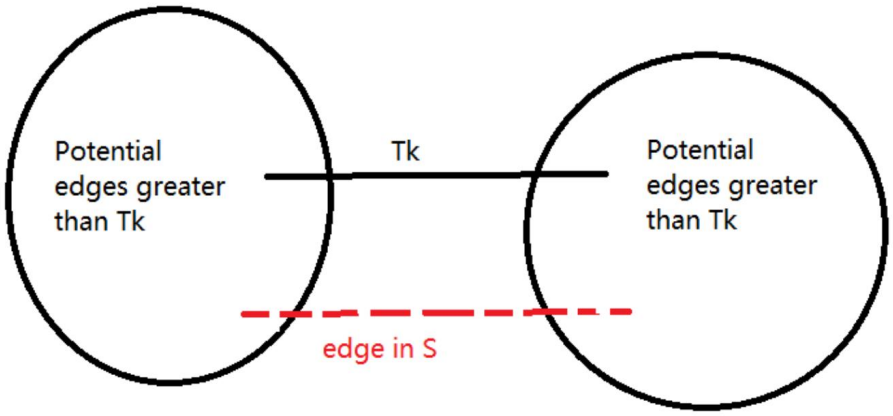
contradiction, and we see that minimum spanning tree cannot have paths from v other than bottleneck paths. Thus all the paths from v to other nodes are bottleneck paths on minimum spanning tree, and minimum spanning tree becomes the same we defined for tree T_k' .

（d）

Now we can compare two spanning trees according to the following criteria: starting from the longest edge, if the longest edges differ, then the spanning tree with the smaller longest edge is better. If the longest path is the same, we compare the second longest path. We keep comparing until the value of the edges differ or we run out of edges. That is, we find in spanning tree A which has edges sorted in decreasing order S1,S2,S3….Sn and spanning tree B which has edges sorted in decreasing order T1,T2,T3….Tn the smallest index k such that Sk is different from Tk, the tree that corresponds to the smaller of Sk and Tk is the better tree.

Now we claim that the spanning tree S that is better than all others according to the above requirement is the minimum spanning tree.

Suppose there is a minimum spanning tree T such that for index j greater than k, Sj = Tj, and for index k, Tk > Sk. According to our criteria, S is better than T. Now, we apply similar argument to part c.



According to the definition of minimum spanning tree, Tk must be the smallest and only edge between the two cuts it separates. If it is not the only edge between the cuts, there will be a cycle and violates MST definition. Thus we know all edges greater than Tk must be in the two cycles. Now consider tree S, all the edges in S that are greater than Tk are the same as all the edges in T that are greater than Tk. But we already know these edges are in the circles and do not connect the two cuts. Thus, the edge in S that connects the two cuts must be smaller than Tk. Now, this generates a contradiction because Tk is no longer the smallest edge between the two cuts. This violates the MST definition and thus this T could not exist. Thus we have proven that T cannot be a minimum spanning tree if T is better than S. And thus the tree S that is better than all other trees is the minimum spanning tree.

3.4 3.d **10 / 10**

✓ **- 0 pts** **Correct**

- 3 pts Need more explanation

- 5 pts Need more explanation

- 10 pts No answer found

4. This problem corresponds to the part in textbook that talk about the implementation of Kruskal's algorithm.

(a)

Initialize an array W of n structs Z1,Z2,Z3…Zn. For index k, node Zk will contain element e_k , an empty pointer, and the set name Sk.

Initialize an array U of length n, For index k, U[k] stores the current size of set k.

Query(e_x, e_y):

follow the pointer in Zx until the pointer becomes empty pointer. Retrieve the set name W1 in the destination struct. Each time the pointer directs us from one struct to another different struct, change the pointer of the old struct to point to struct W1.

follow the pointer in Zy until the pointer becomes empty pointer. Retrieve the set name W2 in the destination struct. Each time the pointer directs us from one struct to another different struct, change the pointer of the old struct to point to struct W2.

If W1=W2:

return e_x, e_y belong to same set

Else:

returns e_x, e_y belongs to different set

Union(e_x, e_y):

follow the pointer in Zx until the pointer becomes empty pointer. Retrieve the set name Sk1. Using the index k1, find the size V1 of the current set in U.

follow the pointer in Zy until the pointer becomes empty pointer. Retrieve the set name Sk2. Using the index k2, find the size V2 of the current set in U.

If Sk1 is not Sk2:

If V1 is smaller:

Make the pointer of SK1 to point to Sk2. V2=V1+V2.

Else:

Make the pointer of SK2 to point to Sk1. V1=V1+V2.

Correctness:

To perform query (e_x, e_y), we first find the set name e_x belongs to, this can be done by following the pointer in Zx until the pointer becomes empty pointer. Retrieve the set name in the destination struct. Then we find the set name e_y belongs to, this can be done by following the pointer in Zy until the pointer becomes empty pointer. Retrieve the set name in the destination struct. If the set names match, then query returns e_x, e_y belongs to same set. Else, return e_x, e_y belong to differentset.

For union (e_x, e_y), we first find the set name e_x and e_y in. Then we change the pointers of each element of the smaller set to point to the bigger set, and increase the size of the bigger set accordingly.

Complexity:

Query will take two find operations which is each O(logN). Now we explain why find

is O(logN): this is justified by the fact that we always combine the smaller set to the larger set, so each time we union the combined set is at least two times as large as the smaller set. The complexity of the find depends on how many times we follow the pointer, each time we follow the pointer two sets are combined and the size of the set (which has the pointer we can follow) is increased by at least two. The maximum size is n, so the maximum number of times we can follow the pointer is O(logN), and thus each find operation is bounded by O(logN). Query will also do extra logN operations on the smaller set to change all the pointers we have followed in the smaller set to point to the larger set. Union operation does two find operation, which is O(logN) complexity, and requires us to change the pointer of the smaller set to point to the larger set, which is only O(1) complexity. and So either Query or Union is bounded by O(logN). To process n such operations, we will need O(NlogN) complexity.

(b)
To implement Kruskal's algorithm, we union the edge with the existing cluster if it does not form a cycle, otherwise we discard the edge.

Sort the edges and put them in array A of structs (the way we defined in the previous part) of length E in increasing order.
Initialize i to be 1.
While i<E:
    If Query(A[i],A[0])= in the same set:
        Do nothing
    Else:
        Union(A[0],A[i])
    i=i+1
return A

Correctness:
This algorithm is the paraphrase of Kruskal's algorithm, if the smallest unvisited edge is from outside the cluster, include it in current cluster, otherwise discard it.

Complexity:
The sorting will take O(ElogE), since E<2V, we can rewrite it as O(E logV). We will do Query E times, each query is bounded by O(logE) so in total it is O(ElongE). The minimum spanning tree will have V-1 edges so we will do at most V-2 union. Because each Union(A[0],A[i]) happens only after Query(A[i],A[0]), the pointer from the edge to its set name can be at most of length 1. Thus each union is O(1) complexity. Thus the total complexity is O(ElogV+V-2)=O(ElogV).

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

is O(logN): this is justified by the fact that we always combine the smaller set to the larger set, so each time we union the combined set is at least two times as large as the smaller set. The complexity of the find depends on how many times we follow the pointer, each time we follow the pointer two sets are combined and the size of the set (which has the pointer we can follow) is increased by at least two. The maximum size is n, so the maximum number of times we can follow the pointer is O(logN), and thus each find operation is bounded by O(logN). Query will also do extra logN operations on the smaller set to change all the pointers we have followed in the smaller set to point to the larger set. Union operation does two find operation, which is O(logN) complexity, and requires us to change the pointer of the smaller set to point to the larger set, which is only O(1) complexity. and So either Query or Union is bounded by O(logN). To process n such operations, we will need O(NlogN) complexity.

(b)
To implement Kruskal's algorithm, we union the edge with the existing cluster if it does not form a cycle, otherwise we discard the edge.

Sort the edges and put them in array A of structs (the way we defined in the previous part) of length E in increasing order.
Initialize i to be 1.
While i<E:
    If Query(A[i],A[0])= in the same set:
        Do nothing
    Else:
        Union(A[0],A[i])
    i=i+1
return A

Correctness:
This algorithm is the paraphrase of Kruskal's algorithm, if the smallest unvisited edge is from outside the cluster, include it in current cluster, otherwise discard it.

Complexity:
The sorting will take O(ElogE), since E<2V, we can rewrite it as O(E logV). We will do Query E times, each query is bounded by O(logE) so in total it is O(ElongE). The minimum spanning tree will have V-1 edges so we will do at most V-2 union. Because each Union(A[0],A[i]) happens only after Query(A[i],A[0]), the pointer from the edge to its set name can be at most of length 1. Thus each union is O(1) complexity. Thus the total complexity is O(ElogV+V-2)=O(ElogV).

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