CS 180 HW3

Hanna Co

Due: May 24, 2021

1.

Once a node is remove from the priority queue, the algorithm assumes that we have already found the shortest path to it. Take the graph with nodes A, B, and C, and edges (A, B, 5), (A, C, 3), (B, C, -5). If we run Dijkstra from A, we first look at A, and it's two outgoing edges. We then have $\pi[B] = 5$ and $\pi[C] = 3$. The next iteration looks at C, and sees that it had no outgoing edges, so it assumes that we have already found the shortest path from A to C, which has weight 3. However, this is not true, because if we take the path $A \to B \to C$, we see that this path has weight 0. So Dijkstra's algorithm fails here.

2.

If $w' < w$

    return true (still a minimum spanning tree)

Else

    Let's remove (u, v) from the MST
    The MST is now split into two components, A and B
    $min = w'$
    For every edge y in the graph:
        if $weight(y) < min$ and y connects A and B
            return false (no longer an MST)
    return true (still an MST)

Proof of correctness:

The first case is fairly self explanatory: decreasing an edge in the MST preserves the properties of an MST. Let's look at the more complex case. Say our algorithm says our tree is still an MST, but there is in fact a shorter way to connect all the nodes. Given our algorithm, we know this is not possible. Separately, A and B are already MSTs. Our algorithm find the shortest edge that connects the two, which guarantees minimum cost. Therefore, out algorithm is correct.

Proof of time complexity:

The first case should be a really simple check, depending on how the minimum spanning tree is stored, but it should not take more than $O(m)$. In the last case, the worst case scenario has us looking at all the edges in the graph, which is also $O(m)$. Thus, this algorithm runs in $O(m)$ time.

3.

Proof:

Let's say that the traveling salesman manages to visit every node with a path length that is less than the MST length. This would mean that he was able to find a path to all the nodes that is shorter than the MST path. However, by definition, a MST has the lowest weight of traveling to any node. Thus, it follows that there is no shorter way to reach any node in the graph, other than the MST path. Therefore, the optimal tour cannot be less than the MST. However, it can be greater, since the traveling salesman may have to return to the same node multiple times.

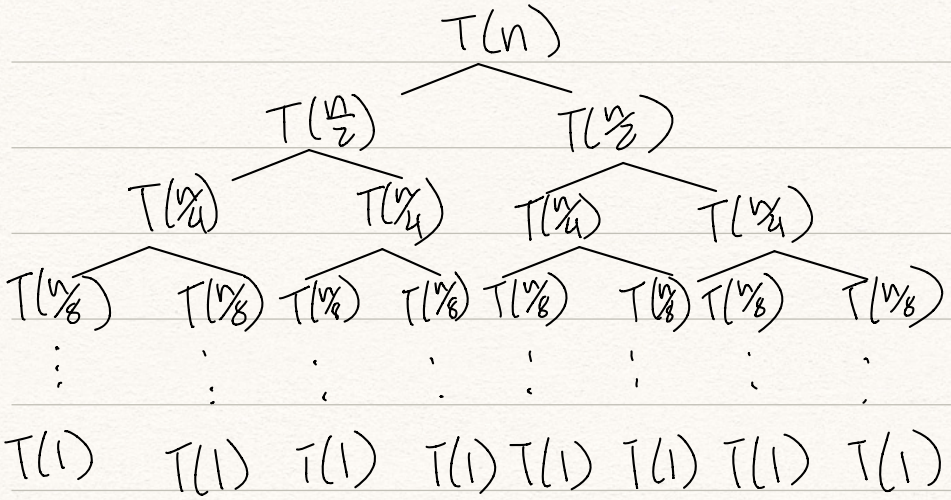
Algorithm:

One possible algorithm is to construct the MST, and then list the nodes in pre-order traversal, adding the starting node at the very end.

Proof of correctness:

Obviously, this path covers all nodes in the graph. In order to visit each node, the salesman will have to walk each "branch", so the weight of their total path will be at least the weight of the MST. The weight of this path is also less than or equal to twice the cost of the MST, because the salesman will not walk any path more than twice. They will walk it once to reach the desired node, and walk it again for the second and final time to "leave" the node. Thus, it follows that the total cost of any path produced by this algorithm is less than or equla to twice the total cost of the MST.

$T(n) = 2T(\frac{n}{2}) + n\log n$



$l=0$, $n\log n$

$l=1$, $2(\frac{n}{2})\log(\frac{n}{2})$

$l=2$, $2^2(\frac{n}{4})\log(\frac{n}{4})$

$l=3$, $2^3(\frac{n}{2^3})\log(\frac{n}{2^3})$

$l=0$: $n\log n$

$l=1$: $n(\log n - 1)$

$l=2$: $n(\log n - 2)$

$l=3$: $n(\log n - 3)$

$\vdots$

$l=i$, $n(\log n - i)$

$n(\log n - \log n) = 0$

$$T(n) = \sum_{i=0}^{\log n - 1} n(\log n - i)$$

$$= n\left[\sum_{i=0}^{\log n - 1} \log n - \sum_{i=0}^{\log n - 1} i\right]$$

$$= n\left[(\log n)^2 - \frac{(\log n - 1)(\log n)}{2}\right]$$

$$= n\log n\left(\log n - \frac{\log n - 1}{2}\right)$$

$$= \frac{1}{2} n\log n(2\log n - \log n - 1)$$

$$T(n) = \frac{n\log n}{2}(\log n - 1)$$

$$T(n) = \frac{n \log^2 n}{2} - \frac{1}{2}$$

$$= O(n \log^2 n)$$

5.
We can solve this problem using divide and conquer

If our array only has one element:
    return $A[0]$
Otherwise:
    We split our array A into two parts, $A[0 \sim \frac{n}{2}]$ and $A[\frac{n}{2} \sim n]$
    We recursively call our algorithm on the two halves
    If they have the same majority element m
        return m
    If one sub-array has a majority element m
        $count = 0$
        for every element e in A:
            if $e = m$
                $count++$
        if $count > \frac{n}{2}$
            return m
    If both sub-arrays have a majority element m and l
        $mcount = 0$
        $lcount = 0$
        for every element e in A:
            if $e = m$
                $mcount++$
            else if $e = l$
                $lcount++$
        if $mcount > \frac{n}{2}$
            return m
        if $lcount > \frac{n}{2}$
            return l
    Else
        return false (no majority element)

Proof of correctness:
Let's say that A does in fact have a majority element m, but our algorithm returns false. This would mean that m is either a majority element in only one half of the array, or not a majority element in either half. Consider the first case: m is a majority element in one of the halves. If our algorithm returns false, this would mean that after iterating through the array, we determined that the number of

occurrences of m in A is less than or equal to $\frac{n}{2}$, meaning that it isn't a majority element in A. Thus, we have a contradiction. In the second case, m is not a majority element in either half of the array. It follows that if the occurence of m in both halves is less than or equal to $\frac{n}{4}$, then the sum of m's occurrences in the entire array must be less than or equal to $\frac{n}{2}$, meaning that m is not a majority element in A, which is also a contradiction. Thus, our algorithm must be correct.

Proof of time complexity:

Since we are halving our array with each iteration, that has a time complexity of $O(log_2 n)$. When checking the array, we iterate over all n elements, which has a time ocmplexity of $O(n)$. So our total time complexity is $O(nlog_2 n)$.