Programming Project INF421 (2023-2024)

Two Centers in a Tree

By MOK Yong and Samuel HON

10 February 2024

Abstract. We introduce a solution to solving the two centers of a weighted tree. With a center defined as the vertices which minimizes the objective function: $S(x, y) = \sum_{v \in V} w(v) \ \cdot \ min\{d(v, x), d(v, y)\}$. More general problems are known to be NP hard (e.g. Finding the k centre of a graph**)**. However, by limiting the problem to 2 centers of a weighted tree, we were able to find an algorithm of O(n) time complexity.

Introduction to the problem. Let *T = (V ,E)* be a tree where *V* is the set of vertices and *E* is the set of edges. For every vertex $v \in V$, it has a weight *w(v)*, which is a positive integer. Denote *d(u,v)* as the distance between u and v in the tree, i.e., the number of edges on the unique path connecting *u* and *v*. If *u* = *v*, then *d(u,v) = 0*. Our task is to find 2 vertices, *x, y* such that we minimize the objective function $S(x, y)$ described in the abstract.

Approach. We first began by simplifying the problem to a path. In which we can easily determine the 2 centers with an O(n) algorithm. Which leads to the question on how we can find the path which contains the 2 centers. If we consider the tree as shown in figure 1. We attempt to determine if C_2 lies on the upper or lower branch. In general, we can apply the same argument to any vertex with multiple branches, which allows us to find the path which contains the 2 centers.

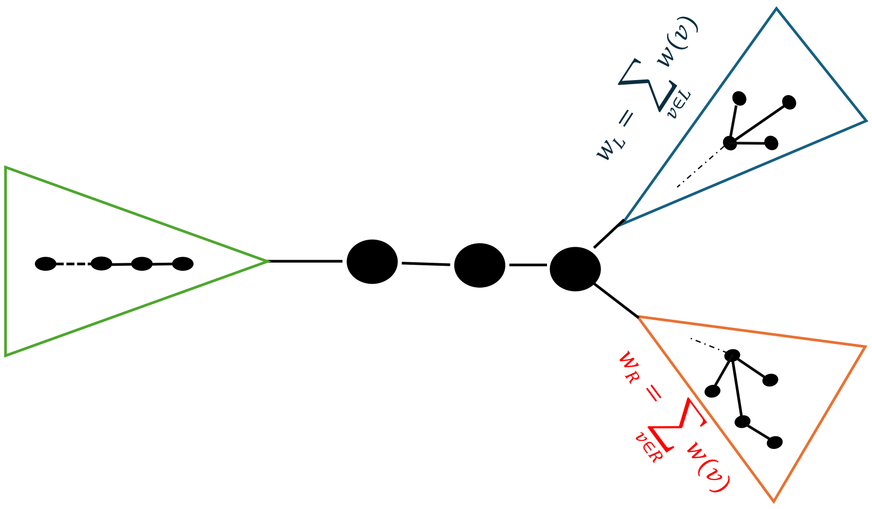


Fig.1. *W_L and W_R is the total weight of left and right sub-tree (In general there are more than two sub-tree).*

Observation. When we move a center from C to C' as in figure 2. We observe that only the sum of the weights on the left (W_L) and right (W_R) affects the total objective function. The distance of the vertices from the end points do not affect the computation of $S(x, y)$.

i.e. $S(C) = S(C') + W_L - W_R$ *(if we are calculating a single center*)*

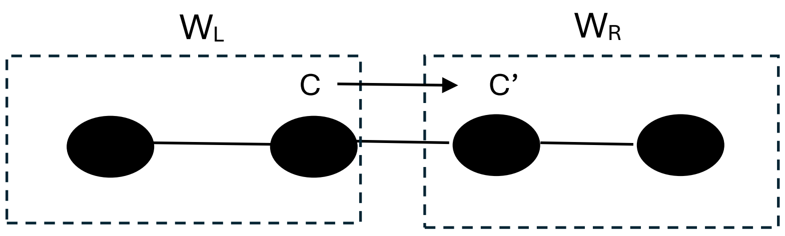


Fig.2. *We consider the weight of left and right tree, denoted by W_L and W_R respectively, if W_R bigger than W_L, we move C to the right and update to C'.*

Approach:

Definition. Minimisability. We define the minimisability of a branch as the

$$depth \times \sum_{v \in Branch} w(v)$$

Where the depth is the distance of the furthest point from the vertex which the branch is connected to.

The motivation here is to have a metric which allows us to determine how much placing a center in the branch can reduce the total objective function. This is appropriate as the deeper the branch, the more the distance metric can be reduced in the objective function. Additionally, as in Fig.2 we see only the sum of the weights matter.

Now suppose that we know that one center is on the left in the green area (Fig.1), and we are trying to find the other center. Intuitively, the second center must lie on the branch on the right which has the highest minimisability. We will prove this later.

Definition. Most minimisable path from *u*. From a vertex *u*, take the branch with the largest minimisability. For the vertex next to *u* on the branch, we repeat the same principle and search for its most minimisable branch (excluding the direction towards *u*). When we reach a leaf, we will have a path. We call this path the most minimisable path from *u*.

The most minimisable path will be defined as the path which maximises the minimisability at each vertex of the path.

This definition is similar, and inspired, by the longest path from *u* definition in the sense that at every vertex, we choose the most minimisable (longest) branch that is connected to it.

Proposition. Computing the most minimisable path in a tree has O(n) time complexity.

Starting from any vertex *u*, find the most minimisable path from *u*, let the vertex at the end of this path be *v.* Now find the most minimisable path from *v*, the resulting path is the most minimisable path. (This is based of the longest path in a tree algorithm)

Proof. Given any starting vertex *u*, suppose that the end point of the most minimisable path from *u* ("calculated" path) is not on the most minimisable path ("true" path). We can then consider the vertex of the "calculated" path which is closest to the "true" path. If this vertex is on the "true" path. Then we have a contradiction at this point, as the algorithm will go along the "true" path instead.

Alternatively, if the closest vertex is not on the "true" path. Then at this point, the branch which contains the "true" path is not as minimisable as branch containing the "calculated" path. Which implies that the "calculated" path is a more minimisable path than the "true" path. (Contradiction)

Now we know that *v* is in the "true" path, and by a similar argument, the algorithm produce a path with an endpoint on the "true" path. Since this is a tree, this path is unique, thus a most minimisable path. *The most minimisable path may not unique.*

This algorithm has a complexity of O(n) as we only need to pass all vertices at most once.

mostMinimisablePath Algorithm (O(n))

```
vertex u
findMinimal (vertex source, vertex u) {
    int totalWeight = u.weight ;
    int depth = 0;
    int minimiseTotal = 0;
    list nextPath = [];
    for (vertex neighbours : vertices adjacent to u) {
        if (neighbours == source) {
            continue;
        }
        neighbourTotal, neighbourDepth, neighbourpath = findMinimal(u, neighbour);
        totalWeight += neighbourTotal;
        if (neighbourTotal * neighbourDepth > minimiseTotal * depth) {
            minimiseTotal = neighbourTotal;
            nextPath = neighbourPath;
            depth = neighbourDepth;
        }
    }
    depth += 1;
    nextPath.add(u)
    return totalWeight, depth, nextPath;
}

cost, minimalPath = findMinimal(u, u)
cost, minimalPath = findMinimal(minimalPath.getFirst, minimalPath.getFirst)
```

Theorem. All most minimisable path contains the 2 centers.

Proof. On a most minimisable path (We refer to as the path for the rest of this proof). Suppose that there is a branch which is not on the path which contains a center, *c*. Fix *v* the vertex closest to *c* which is on the path. Note that *v* cannot be a leaf, if not, the branch will contain the path. Let *b* be the branch of *v* which contains the center (not on the path). Since *v* is on the path, 2 of its branches are on the path, and at least one of these 2 does not contain the other center. From the observation we've made, we realise that if this is the case, we can move the center, *c,* to this branch which does not contain a center, resulting in a smaller output to the objective function, contradicting that *c* is a center. If the branches have the same minimisability, the resulting output of the objective function does not change, because placing a center on either branch is equally minimisable.

Following our algorithm to find the most minimisable path, we see that initially whichever point we start on, we will pass by one of the centers by the argument above. Along the way, if we pass another center. When we repeat the most minimisable path algorithm, we will pass both centers.

If we only passed one center, when we repeat the most minimisable path algorithm, we will reach the center which we've passed. At this point, we can apply the same argument, and see that the algorithm will choose a most minimisable path, which implies that we will pick up the other center.

Having the longest path now, we can prune the tree to the path. With the observation made earlier, we can treat each vertex on a most minimisable path as a new vertex, with a weight equal to the sum of vertices in the branches which are pruned (see Fig.3). In the pruning, we note that we only require O(n) time to condense the vertices not on the most minimisable path.

Proposition. The two centers of a condensed tree to the most minimisable path (see Fig.3) is the same 2 centers of the tree.

Proof. Since we know that the centers lie on the most minimisable path. Based on the observation we made at the start (Fig.2) we know that moving will only add or decrease the objective function by 1 time the sum of weights. Their distances from the path is thus not important, as the distance to the center is always non-zero.

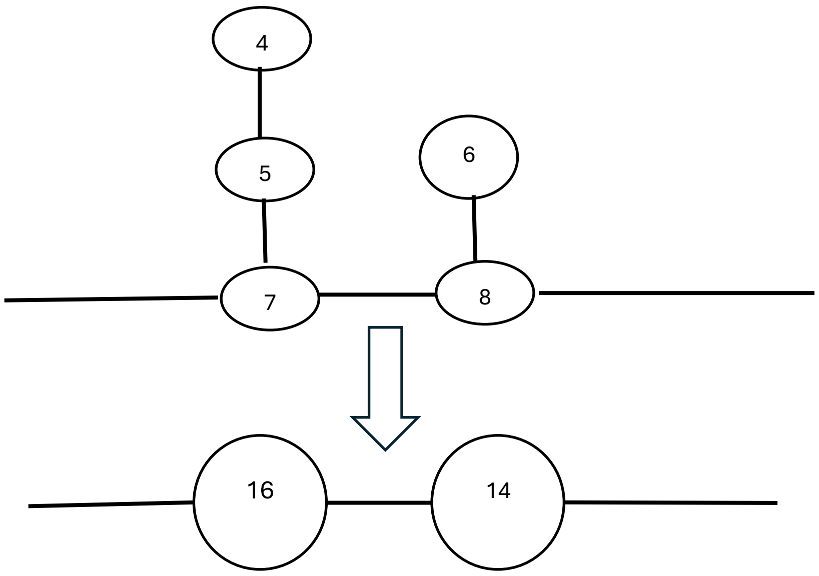


Fig.3 *Condensing a tree into the most minimisable path.*

We finally conclude by finding the two centers on a path.

Proposition. Finding the 2 centers of a path has O(n) time complexity.

Given a path, let C_1 and C_2 be the true centers and P_1 and P_2 be our guess. We begin by placing both P_1 and P_2 at one end of the path (see Fig.4.1).

We initalise the following values

- P_{1l} = weight(P_1)
- P_{1r} = 0
- P_{2l} = weight(P_2)
- P_{2r} = $\sum_{v \in left\ of\ P_2} weight(v)$
- split = P_1 //the furthest vertex from P_1

Note that when we move P_2 right, $S(P_1, P_{2\,new}) = S(P_1, P_{2\,old}) + P_{2l} - P_{2r}$

From this we will move P_2 to the right as long as $P_{2r} > P_{2l}$. After moving, we update the values as follows P_{2l} += weight($P_{2\,new}$), P_{2r} -= weight($P_{2\,new}$). When the distance between the two Ps is odd, we update the split to the vertex on the right of the current split, and P_{1r} += weight(split), P_{2l} -= weight(split).

We update P1 with similar conditions, moving the split point as necessary. More details are provided in the pseudo code.

We now prove that P_1 and P_2 will traverse and find both centers. In total we have 6 cases as shown in figure 4.

Case 1 (Fig.4.1) Since the centers are on the right of P_2 , we can always move P_2 to the right, and find a better solution.

Case 2 (Fig.4.2) P_2 is the nearer center. This could actually cause the algorithm to stop as this point might be so massive that moving to the left or right of it is not optimal. However, by simply repeating the algorithm with P_1 and P_2 on the opposite end of the path, we can see that P_1 in this case will not get stuck at the center on the right. *If not P_2 would not have gotten stuck at the left center in the first place*. As such we can simply run the algorithm twice from opposite directions and compare the results. This will only double the time complexity.

Case 3 (Fig.4.3) P_2 is between the centers and P_1 is not. This could also pose a problem, however, as in case 2, if we run the algorithm in the opposite direction, we will see that P_1 will not pass the left center. Which we prove in case 5.

Case 4 (Fig.4.4) P_2 stops at the right center. Here, clearly P_1 continues to move, as it will not be optimal.

Case 5 (Fig.4.5) P_2 passes the right center, while P_1 remains on the left of the left center. This is not possible as at some point, the vertices closer to P_2 (*than P_1*) would be a superset of the vertices closer to the left center (*than the right center*), and since the left center is optimal, P_2 would not move any more to the right.

Case 6 (Fig.4.6) A symmetric argument to case 5 would also show that P_1 cannot move past the left center.

As such we need to run this search algorithm twice, in opposite directions. We can then take the total value of each pair of centers, compare them, and find the lower of the 2.

The time complexity of each search is at most 3n as each of the 2 P will only move at most n vertices + n for the initialisation. Since we run the search twice, we get a time complexity of 6n or O(n).

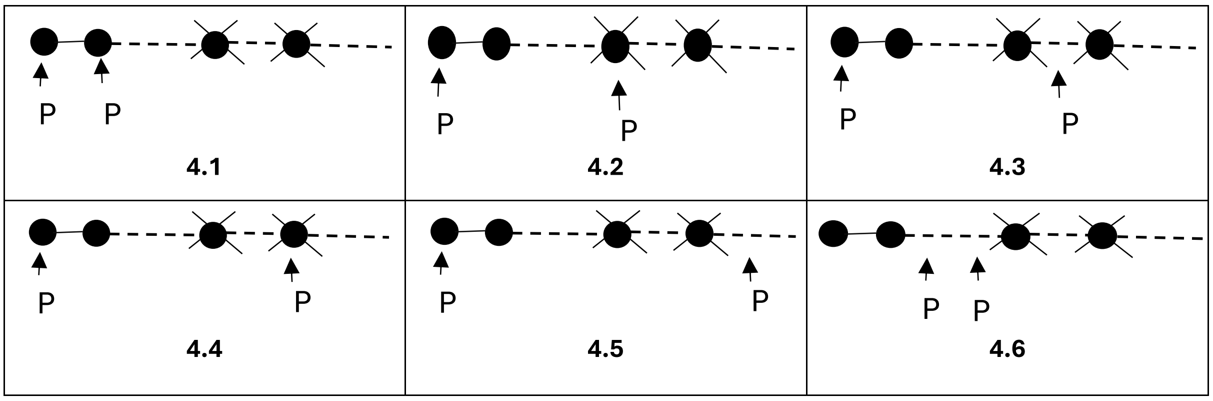


Figure 4. *Cases described when searching for the 2 centers along the path.*

findCenters

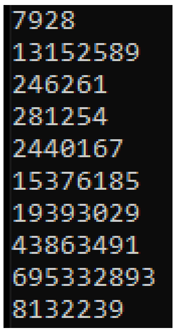
```
**Path** path;
$P_1$, $P_2$ = 0, 1; //using index here instead of the vertices
int c1l = weight(path.get($P_1$))
int c1r = 0
int c2l = weight(path.get($P_2$))
int c2r = $\sum_{v \in left\ of\ P_2} weight(v)$ //can be done in O(n) time

boolean go = true;
**while (**go**)** {
    go = false;
    **if (**c2r > c2l**)** {
        go = true;
        $P_2$ += 1;
        c2r -= weight(path.get($P_2$));
        c2l += weight(path.get($P_2$));
    }
    **if (**c1r > c1l**)** {
        go = true;
        $P_1$ += 1;
        c1r -= weight(path.get($P_1$));
        c1l += weight(path.get($P_1$));
    }
    **if (**($P_2$ – $P_1$ ) % 2) == 1**)** {
        **if** (split >= ($P_2$ – $P_1$)/2 + $P_1$) {
                continue;
        }
        go = true;
        split += 1;
        c2l -= weight(path.get(split));
        c1r += weight(path.get(split));
    }
}
```

Final Solution. Given a tree, begin by finding a most minimisable path. This path will contain the 2 centers. Next, we condense the tree to the most minimisable path, which we can then find the 2 centers of. After we found the 2 centers, we find the center edge between the two centers on the original tree (*if there 2, remove 1 of the 2, the calculation will still be the same*), and remove this edge from the original tree. Finally, we can perform a pre-order traversal of the 2 trees (breath first search) to calculate the objective function $S(x, y)$.

Note that, finding the most minimisable path costs O(n), pruning the tree to get a modified path costs O(n), finding the 2 centers costs O(n) and a breath first search will also cost O(n) time (*if implemented properly*). The total time complexity of our solution is O(n). A pseudo code for a pre-order traversal of the 2 trees in O(n) time is presented in the annex.

The result of the algorithm on the test codes are as shown in figure 5.

To run the code: $ java Center



```
7928
13152589
246261
281254
2440167
15376185
19393029
43863491
695332893
8132239
```

***Fig. 5**, results of algorithm on test cases in order from 1 to 10.*

Conclusion.

In this problem we have found an algorithm which finds the two centers of a tree with a time complexity of O(n)

We heavily rely on the fact that the 2 centers lie on the most minimisable path. As such generalising our solution to k-centers is not ideal since the 3 centers might not necessarily lie on a path. However, there could be potential for finding the 2 centers in graphs by first finding a path, however, this is known to be NP hard.

We also found an alternative algorithm with a complexity of O(n^2).

By cutting an edge, we will form 2 trees, from here we can find the center of each tree O(n) and compute the score O(n). Repeating this for all the edges, we can find the two centers which minimises the objective function.

Annex.

calculateTotal O(n)

```
Tree t;
vertex center1, center2;

Edge middle = findMiddleEdge(center1, center2) //Depth First Seach from center1 to center2);
t.removeEdge(middle)

int checked = 0;
int depth = 0;
list toCheck = [center1, center2] ;
int total = 0;
while (checked < number of vertices) {
    list Buffer = []
    while (toCheck is not empty) {
        checking = first element of toCheck;
        remove checking from toCheck;
        add all neighbours of checking to buffer;
        total += weight(checking) * depth;
        checked += 1;
        remove edge adjacent to checking;
    }
    Add all of buffer to toCheck
    depth += 1;
}
return total
```