Two Centers in a Tree

INF 421 - Assigment 5

Project proposed by Hang Zhou, professor at Ecole Polytechnique

Proposed solution in Java by
Diane BERANGER and Thibaut VAILLANT

3 February 2020

Contents

1 Reminder : Problem Description

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 $d(u, v) = 0$.

Your task is to find two vertices x and y in the tree, such that the following expression $S(x, y)$ is minimized:

$$S(x, y) = \sum_{v \in V} (w(v) \times min\{d(v, x), d(v, y)\})$$

Such vertices x and y are the two centers in the tree.

2 Analysis of the problem

The aim in this project is to find two centers in a tree where vertices have positive weights (2-center problem).

It is a particular case of the p-center problem, which consists in finding p centers in a graph where vertices and edges hold positive weights (a possible application of this is the determination of optimal position of warehouses, considering in their distances to different cities, and the amount of products each of these cities ordered).

The problem of a 1-center in a **tree** can be solved efficiently with Goldman's algorithm in time $O(N)$. Below are the steps of Goldman's algorithm (we will use this algorithm to compute the 2-centers in our "LinkDeletion" algorithm) :

- Step 1: If T consists in a single vertex then stop ; this vertex is the 1-center;
- Step 2: Search for an end-vertex v_1. If $w(v) \geq w(T)/2$, go to step 4; otherwise go to step 3;
- Step 3: Let v_2 be the adjacent vertex of v_1. Modify T by deleting the edge $[v_1, v_2]$ and incremeting $w(v_2)$ by $w(v_1)$. Then return to Step 1;
- Step 4: Stop : v_1 is a 1-center.

Let us go back to our 2-center problem. A naive approach (for instance, computing a distance matrix using Floyd-Warshall algorithm, and then determining $S(x, y)$ for each $x, y \in V$) takes time $O(N^3)$. We implemented this naive solution and found out that it worked for test trees 1 to 4, but was too slow for the bigger test trees. We therefore had to find more time-efficient solutions.

In the following sections you will find a description and complexity analysis of the two main solutions we propose : the first one (a greedy algorithm) finds

approximate solutions but is extremely time efficient (runs in at most a couple of seconds on the test trees) ; the second one (the "LinkDeletion" algorithm) computes the exact solution but is way slower.

3 Greedy algorithm

This algorithm provides, in some test cases, the exact solution, and in some others (4,8,9,10), a very good approximation of it. We chose to include it in this report even if it is not correct, because in some practical cases, obtaining a good approximation of the solution in reasonable time be useful.

For this algorithmic solution, we will use two classes:

- the class "Tree", in which we will store the edges and vertices of our tree, as well as non-static method "adj_lists" that computes lists of neighbors for all vertices
- the class "Greedy" which computes the two centers of the tree, by updating static variables c_1 and c_2 step by step (following a rule that we describe below). The tree we compute the centers of is a static variable of "TwoCenters" too, as well as the score of the current centers, $S(c_1, c_2)$

Here are the steps followed by our algorithm in the "main" method of the class "TwoCenters" :

1. **Step 1: Read the file**

   We store the edges in an "edge" array of length $N-1$, each element of which is a 2 elements array storing an edge $[i, j]$.

   We store the weights of the vertices in an array of length N, named "w".

2. **Step 2:** Store the edges in the form of adjacency lists (more practical)

   We do this in time $O(n)$.

3. **Step 3: Find the diameter of the tree**

   By diameter of the tree, we mean the longest path in the tree. It can be shown (clear with a drawing) that if we take any end-vertex (root) v_0 and find a vertex v_1 which is the farthest away from v_0, and then find a vertex v_2 which is the farthest away from v_1, then v_1-v_2 is a diameter of the tree.

   We choose arbitrarily $v_0 = 0$ (always an end-vertex in the tests), then we do :

   - one BFS (breadth first search) to find v_1
   - another BFS to find v_2

- a DFS (depth first search) to find the path $v_1 - v_2$ (having this path is useful to compute the score S)

4. **Step 4: Starting with c_1 and c_2 in the middle of the diameter, move them away from each other step by step in the way that decreases the score the most**

   At each iteration of the "while" loop, we calculate the score of each couple (x, y) where x is c_1 itself or one of its neighbors, and y is c_2 or one of its neighbors. The (x,y) with the lowest score becomes the new (c_1, c_2).

   When it is no more possible to improve the score by moving the centers this way, we exit the "while" loop.

Here are the time-complexities of all steps above :

1. **Step 1:**

   This is done in time $O(n)$

2. **Step 2:**

   Done in time $O(N)$ as well.

3. **Step 3:**

   BFS and DFS have a $O(N)$ time complexity, so step 3 is in $O(N)$ as well.

4. **Step 4:**

   This is the step with the highest time complexity.

   Each update involves two BFS, so its time complexity is in $O(N)$.

   There cannot be more than N updates (one of the centers moves at each update, and we cannot go backwards).

   Therefore this gives us a $O(N^2)$ time-complexity.

To conclude, the worst-case complexity of our "Greedy" algorithm is bounded above by $C \times N^2$. In practice, for the trees in the test set we are given, the runtime increases rather linearly (see section 5).

4 LinkDeletion algorithm

For this algorithmic solution, we will use two classes:

- the class "Tree", the same as the one we used for the "Greedy" algorithm
- the class "LinkDeletion" (the algorithm itself)

This algorithm finds the two (exact, contrary to "Greedy") centers of the tree. To do so, after reading the test file and storing the edges in adjacency lists (the same way as in the greedy algorithm), "LinkDeletion" executes the following steps $N-1$ times (once for each edge e in the tree) :

1. **Step 1:**

   We partition the tree T into two subtrees T_1 and T_2 located on both sides of edge e. Let V_1 (resp. V_2) be the set of vertices of T_1 (resp. T_2). We delete (temporarily : for this set of steps, until we examine the next partition) the edge e to separate the two subtrees.

2. **Step 2:**

   Using Goldman's algorithm (we enumerated the steps of Goldman's algorithm in Section 2 - Analysis), we find c_1 and c_2, the 1-centers (ie those who minimize the weighted sum of distances in the subtree) of T_1 and T_2. This step takes time $O(N)$, because Goldman's algorithm has linear time-complexity (it deletes one vertex at each iteration).

3. **Step 3:**

   We compute J for this partition, where J is:

   $$J(T_1, T_2) = S_1(c_1) + S_2(c_2) = \sum_{v_1 \in V_1} w(v_1)\, d(v_1, c_1) + \sum_{v_2 \in V_2} w(v_2)\, d(v_2, c_2)$$

   (S_1 and S_2 are the score functions in the space of subtrees T_1 or T_2, same as S which is the score function in the space of tree T)

   With BFS, computing J is done in time $O(N)$.

The c_1 and c_2 of the partition that minimizes J are the 2-centers of the tree T.

We repeat $N-1$ times the set of steps {1,2,3}, which has complexity $O(N)$. Thus the total time-complexity of this link-deletion algorithm is $O(N^2)$.

5 Outputs and runtimes for both algorithms

Algorithm	**LinkDeletion (exact solutions)**				Greedy (approximate solutions)			
Test	c_1	c_2	Score	Runtime	c_1	c_2	Score	Runtime
1	9	32	7928	121247300	9	32	7928	40194100
2	9	5	246261	6994435900	9	5	246261	144046500
3	8	4	281254	13124793100	8	4	281254	180093600
4	15	77	2440167	30459982800	12	61	2661508	253828800
5	26	25	15376185	44182792600	26	25	15376185	216026400
6	21	166	19393029	67339738900	21	166	19393029	424340600
7	11	9	43863491	51009671300	11	9	43863491	276820700
8	12	85	695332893	148364497400	85	20	698887909	405587900
9	4	136	8132239	2580282299300	5	7	8324878	568091300
10	15	44	13152589	*	26	13	13201241	838191000

Fig. 1: Array of the outputs and runtimes for both algorithm

The "LinkDeletion" solutions are exact (but test 10 runs in more than an hour), while the solutions given by "Greedy" are merely approximate (however they are computed in less than a couple of seconds).

We plotted the runtimes with Python for both algorithms (with redimensioned x and y axis). Below are the results :

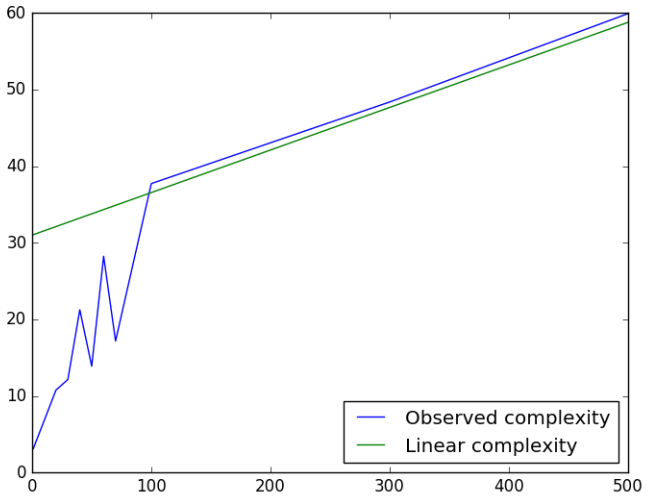

Fig. 2 : Runtime of algorithm "Greedy" plotted against $31 + N/18$

We notice that the complexity of algorithm "Greedy" is linear rather than quadratic when N is big enough ($N = 10000, 30000, 50000$). We can understand this by noticing that the two centers are always located very close to each other in our test trees. Thus the number of iterations of the while loop in "Greedy" (each iteration of which is in time $O(N)$ is in fact bounded above, hence a linear complexity.

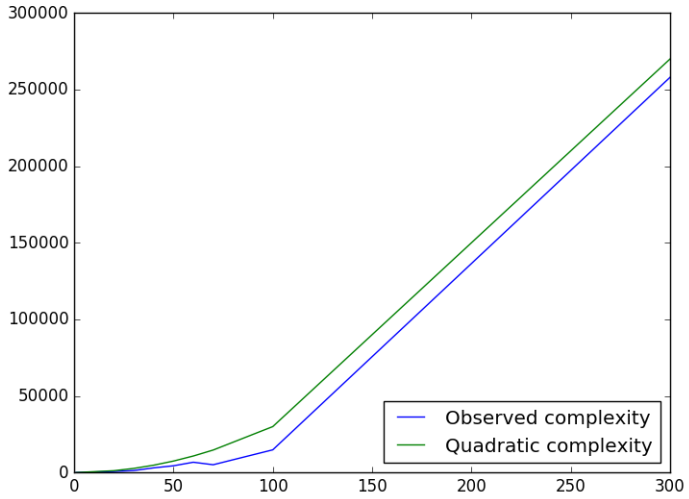

Fig. 3 : Runtime of algorithm "LinkDeletion" plotted against $3 \times N^2$

The observed runtimes of algorithm "LinkDeletion" are compatible with our theoretical quadratic complexity.