Optimal algorithm for computing Steiner 3-eccentricities of trees

Aleksandar Ilić
Facebook Inc, Menlo Park 94025, California, USA
e-mail: aleksandari@gmail.com

August 24, 2020

Abstract

The Steiner $k$-eccentricity of a vertex $v$ of a graph $G$ is the maximum Steiner distance over all $k$-subsets of $V(G)$ which contain $v$. In this note, we design a linear algorithm for computing the Steiner 3-eccentricities and the connective Steiner 3-eccentricity index on a tree and thus improving a quadratic algorithm presented in [G. Yu, X. Li, Connective Steiner 3-eccentricity index and network similarity measure, Appl. Math. Comput. 386 (2020), 125446.]

Key words: Linear algorithm; Steiner distance of trees; Connective Steiner 3-eccentricity index.

1 Introduction

Let $G = (V, E)$ be a simple connected and undirected graph with vertex set $V(G)$ and weighted edge set $E(G)$. The distance $d(u,v)$ between two vertices $u$ and $v$ in $G$ is the length of a shortest path connecting $u$ and $v$. The eccentricity $ecc(v)$ of a vertex $v$ is the maximal distance between $v$ and any other vertex.

The Steiner distance was introduced by Chartrand, Oellermann, Tian and Zou [2] in 1989 as a generalization of general distance. For a set $S \subseteq V(G)$ of $k \geq 2$ vertices, a Steiner tree of $S$ is the minimum weight connected subgraph of $G$ which contains all vertices of $S$. The Steiner distance $SD(S)$ among the vertices $S$ is the minimum weight among all connected subgraphs whose vertex sets contain $S$. Therefore,

$$SD(S) = \min \left\{ \sum_{e \in E(T)} w(e) : T \text{ is a subtree of } G, S \subseteq V(T) \right\},$$

where $T$ is a Steiner tree of $S$, and $w(e)$ represents the weight of edge $e$. The Steiner $k$-eccentricity $\epsilon_k(v)$ of a vertex $v$ of $G$ is defined by

$$\epsilon_k(v) = \max \{SD(S) : S \subseteq V(G), |S| = k, v \in S\}.$$

The classical eccentricity is a Steiner 2-eccentricity, as the distance also equals to the minimum size of a connected subgraph containing these two vertices.

The connective Steiner $k$-eccentricity index $ecc^c_k$ of a graph $G$ is defined by [3]:

$$ecc^c_k = \sum_{v \in V} \frac{\deg(v)}{\epsilon_k(v)} = \sum_{ij \in E} \left( \frac{1}{\epsilon_k(i)} + \frac{1}{\epsilon_k(j)} \right).$$
Yu and Li in [1] designed an $O(n^2)$ polynomial time algorithm to calculate the connective Steiner 3-eccentricity index of trees. Here we present a linear algorithm for calculating the 3-eccentricities for every vertex of weighted trees, and thus improving the above result.

2 Algorithm

Let $T$ be a rooted weighted tree. By $T_r$ denote the subtree rooted at vertex $r \in V$, which is a subgraph induced on vertex $r$ and all of its descendants.

In this section we design $O(n)$ algorithm for computing 3-eccentricities for all vertices of a weighted tree $T$, where $n$ is the number of vertices in $T$.

First we choose an arbitrary vertex as the root of $T$. The adjacency list $adj$ is used to store the weights of the edges. The main idea is to run two-stage tree traversals using depth first search procedure (see [4] for details). In the first stage, for every vertex $v$ we compute the longest paths from $v$ in the subtree rooted at $v$. In the second stage we update the longest paths with the upwards path via parent node in order to compute 3-eccentricities.

In each recursive call we maintain three vectors:

- $\text{path\_weight}[v]$ representing the length of the longest path from the vertex $v$ in the subtree $T_v$
- $\text{path\_index}[v]$ representing the neighbor of $v$ on the longest path from the vertex $v$ in the subtree $T_v$
- $\text{attached\_weight}[v]$ representing the length of the longest subpath attached at the longest path in the subtree $T_v$ strictly below $v$

In fact, we need to store the top three longest paths and corresponding indexes and attached paths for every vertex in order to have efficient implementation. Finally, 3-eccentricity of the vertex $v$ can be computed as

$$
\epsilon_3(v) = \text{path\_weight}[v][0] + \max\{\text{path\_weight}[v][1], \text{attached\_weight}[v][0]\}.
$$

We also need to store two more vectors:

- $\text{parent}[v]$ representing the unique parent of $v$ in the DFS tree (for the root vertex, we have $\text{parent}[v] = -1$)
- $\text{mark}[v]$ representing marked nodes in dfs tree

The main helper function is $\text{update}(v, u, new\_weight, new\_attached\_weight)$ and it serves to update the above vectors and keep the top three values for the longest paths coming from the vertex $v$ (see Appendix for the implementation).

The first stage $\text{DFS\_stage1}$ is composed of simple initialization of the vectors and then recursively traversing all neighbors of the vertex $v$ (that are not the unique parent of $v$). After the subtree is processed with the recursive call, we update the values for the node $v$ based on the values for each of the subtrees. Note that for leaves the downstream values for $\text{path\_weight}[v]$ and $\text{attached\_weight}[v]$ can be initialized to 0.

In the second stage $\text{DFS\_stage2}$, we traverse the nodes in the pre-order phase: first we update the values for the root and then run computation for the neighbors. For updating the values for
Algorithm 1: DFS\(_{\text{stage1}}(v)\)

**Input:** The adjacency matrix of the tree \(T\) with the root vertex \(\text{root}\).

**Output:** The downwards arrays \(\text{path\_weight}, \text{path\_index}, \text{attached\_weight}\).

\[
\text{path\_weight}[v] = (0, 0, 0);
\text{path\_index}[v] = (-1, -1, -1);
\text{attached\_weight}[v] = (0, 0, 0);
\]

for every neighbor \(u\) of \(v\) do
  if \(\text{parent}[u] = -1\) and \(u \neq \text{root}\) then
    \(\text{parent}[u] = v;\)
    \(\text{DFS\_stage1}(u);\)
    \(\text{update}(v, u, \text{adj\_weight}[v][u] + \text{path\_weight}[0][u], \max(\text{path\_weight}[1][u], \text{attached\_weight}[0][u]));\)
  end
end

Algorithm 2: DFS\(_{\text{stage2}}(v)\)

**Input:** The outputs of \(\text{DFS\_stage1}\).

**Output:** The arrays \(\text{path\_weight}, \text{path\_index}, \text{attached\_weight}\).

\(\text{mark}[v] = 1;\)
\(u = \text{parent}[v];\)
if \(u \neq -1\) then
  \(\text{up\_path\_weight} = 0;\)
  \(\text{up\_attached\_weight} = 0;\)
  if \(\text{path\_index}[0][u] \neq v\) then
    \(\text{up\_path\_weight} = \text{adj\_weight}[u][v] + \text{path\_weight}[0][u];\)
    if \(\text{path\_index}[1][u] \neq v\) then
      \(\text{up\_attached\_weight} = \max(\text{path\_weight}[1][u], \text{attached\_weight}[0][u]);\)
    end
  else
    \(\text{up\_attached\_weight} = \max(\text{path\_weight}[2][u], \text{attached\_weight}[0][u]);\)
  end
else
  \(\text{up\_path\_weight} = \text{adj\_weight}[u][v] + \text{path\_weight}[1][u];\)
  \(\text{up\_attached\_weight} = \max(\text{path\_weight}[2][u], \text{attached\_weight}[1][u]);\)
end
\(\text{update}(v, u, \text{up\_path\_weight}, \text{up\_attached\_weight});\)
end
for every neighbor \(u\) of \(v\) do
  if \(\text{mark}[u] = -1\) then
    \(\text{DFS\_stage2}(u);\)
  end
end
the parent node $u$, consider three cases: if the longest path from $u$ goes through the vertex $v$, if the second longest path from $u$ goes through the vertex $u$, or otherwise if $\text{path\_index}[u][0] \neq v$ and $\text{path\_index}[u][1] \neq v$. See Appendix for the details of the implementation.

The correctness of the algorithm directly follows from the definition of 3-eccentricities and given that we traverse the vertices twice - the time and memory complexity of the algorithm is $O(n)$.

3 Appendix

```c
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>

using namespace std;
#define MAXN 1000

int n, root;
int deg[MAXN];
int adj_list[MAXN][[MAXN];
int adj_weight[MAXN][[MAXN];
int parent[MAXN];
int mark[MAXN];
int path_weight[3][MAXN];
int path_index[3][MAXN];
int attached_weight[3][MAXN];
int steiner3[MAXN];

void update (int v, int u, int new_weight, int new_attached_weight) {
    int index = -1;
    if (path_weight [0][v] <= new_weight) {
        path_weight [2][v] = path_weight [1][v];
        path_index [2][v] = path_index [1][v];
        attached_weight [2][v] = attached_weight [1][v];
        path_weight [1][v] = path_weight [0][v];
        path_index [1][v] = path_index [0][v];
        attached_weight [1][v] = attached_weight [0][v];
        index = 0;
    } else if (path_weight [1][v] <= new_weight) {
        path_weight [2][v] = path_weight [1][v];
        path_index [2][v] = path_index [1][v];
        attached_weight [2][v] = attached_weight [1][v];
        index = 1;
    } else if (path_weight [2][v] < new_weight) {
        index = 2;
    }
    if (index > -1) {
        path_weight [index][v] = new_weight;
        path_index [index][v] = u;
        attached_weight [index][v] = new_attached_weight;
    }
}

void dfs_stage1 (int v) {
    path_weight [0][v] = 0;
    path_weight [1][v] = 0;
}
for (int i = 0; i < deg[v]; i++) {
    int u = adj_list[v][i];
    int weight = adj_weight[v][u];
    if ((parent[u] == -1) && (u != root)) {
        parent[u] = v;
        dfs_stage1(u);
        update(v, u, weight + path_weight[0][u],
              max(path_weight[1][u], attached_weight[0][u]));
    }
}

void dfs_stage2(int v) {
    mark[v] = 1;
    int u = parent[v];
    if (u != -1) {
        int up_path_weight = 0;
        int up_attached_weight = 0;
        int weight = adj_weight[u][v];
        if (path_index[0][u] != v) {
            up_path_weight = weight + path_weight[0][u];
            if (path_index[1][u] != v) {
                up_attached_weight = max(path_weight[1][u], attached_weight[0][u]);
            } else {
                up_attached_weight = max(path_weight[2][u], attached_weight[0][u]);
            }
        } else {
            up_path_weight = weight + path_weight[1][u];
            up_attached_weight = max(path_weight[2][u], attached_weight[1][u]);
        }
        update(v, u, up_path_weight, up_attached_weight);
    }
    for (int i = 0; i < deg[v]; i++) {
        int u = adj_list[v][i];
        if ((mark[u] == -1) && (u != root)) {
            dfs_stage2(u);
        }
    }
}

int main() {
    construct_tree();

    for (int i = 0; i < n; i++) {
        parent[i] = -1;
        mark[i] = -1;
    }
    root = 0;
    dfs_stage1(root);
    dfs_stage2(root);
for (int i = 0; i < n; i++) {
    steiner3[i] = path_weight[0][i] + max(path_weight[1][i], attached_weight[0][i]);
}
return 0;

References

[1] G. Yu, X. Li, Connective Steiner 3-eccentricity index and network similarity measure, Appl. Math. Comput. 386 (2020), 125446.

[2] G. Chartrand, O.R. Oellermann, S. Tian, H.B. Zou, Steiner distance in graphs, Časopis pro pestování matematiky 114 (1989) 399–410.

[3] K. Xu, K. Ch. Das, H. Liu, Some extremal results on the connective eccentricity index of graphs, J. Math. Anal. Appl. 433 (2016) 803–817.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.