CREATIVE RESEARCH

KOREA SCIENCE ACADEMY OF KAIST

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

An Efficient Algorithm for Finding the Eccentricity Sequence of Cacti Based on Trees

Author: *23-035 Changha Kim*
23-031 Junee Kim
23-002 Junkyu Kang
Instructor: *Jerome Tambour*

1 Introduction

Eccentricity sequence is one of the great data to represent graphs. Since we can get the eccentricity of some vertex by running a graph traversal algorithm, it is well-known to compute the eccentricity sequence in $O(|V|^2)$ time where $|V|$ is the number of vertices. However, the eccentricity sequence on some kinds of graphs could be found in lower time complexity. For instance, an algorithm for computing the eccentricity sequence of trees, which uses $O(|V|)$ time, is already known. Based on trees, we created an efficient method to decrease the time complexity of finding the eccentricity sequence for cacti, graphs that do not have any edges shared by two different cycles. By the method we found, the eccentricity sequence is obtainable in $O(|\Gamma||V|)$ time complexity, where $|\Gamma|$ is the number of cycles in a graph.

2 Basics of Graphs

Here, we only discuss undirected, unweighted, and simple graphs.

Definition 1. *For a graph $G = (V, E)$, the degree of some vertex v, denoted as $\deg_G(v)$, is the number of edges connected to v.*

Lemma 2. ***Handshaking Lemma**: For any graph $G = (V, E)$, the following holds.*

$$\sum_{v \in V} \deg_G(v) = 2|E|$$

Definition 3. *A path is a sequence of distinct vertices $(v_1, v_2, \cdots, v_n)$ where there exists an edge connecting v_i and v_{i+1} for every $i = 1, 2, \cdots, n-1$.*

Here, we call v_1 and v_n the *initial vertex* and the *terminal vertex* of the path, respectively.

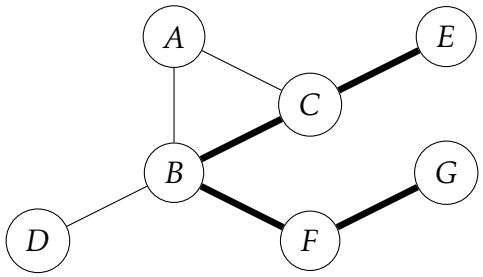


Figure 1

Figure 1 shows a valid path (E, C, B, F, G) in bold. The initial and terminal vertices are E and G, respectively.

Definition 4. *The cycle refers to a closed path in a graph, which means it starts and ends at the same vertex and does not visit any other vertex more than once (except for the initial and terminal vertex). In other words, a cycle is a sequence of vertices and edges that form a loop within the graph.*

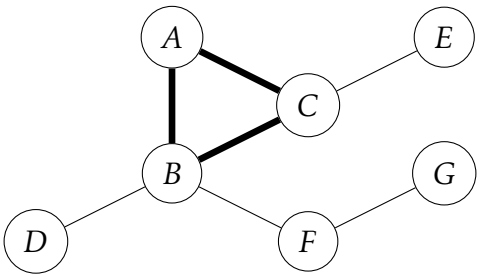


Figure 2

In Figure 2, the portions of the graph that conform to the definition described above are vertices A, B, and C. The cycle they form start at one initial vertex and terminal at the initial vertex again and do not pass the other vertex.

Definition 5. *The distance between two different vertices u and v, denoted as $d(u,v)$, is defined as the length of the shortest path having u as the initial vertex and v as the terminal vertex. The distance between two same vertices is* 0.

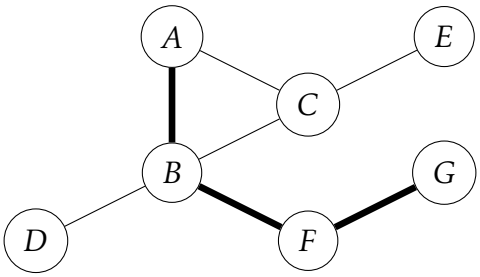


Figure 3

In Figure 3, the distance between vertices A and G is 3. Although there exists two paths (A,B,F,G) and (A,C,B,F,G), since the former one is shorter, $d(A,G) = 3$.

The distance satisfies the following properties where u, v, and w are arbitrary vertices.

- $d(u,u) = 0$
- $d(u,v) = d(v,u)$
- $d(u,v) \leq d(u,w) + d(w,v)$

Definition 6. *The eccentricity of a vertex v is the maximum distance between v and the other vertices u.*

Definition 7. *The radius of a graph is the minimum eccentricity of the graph's vertices.*

Definition 8. *The diameter of a graph is the maximum eccentricity of the graph's vertices.*

Definition 9. *The eccentricity sequence of a graph is the sequence of eccentricity for all vertices in the graph in non-decreasing order.*

In other words, we get the eccentricity sequence by arranging the elements of M in non-decreasing order.

$$M = \left\{ \max_{v \in V} d(u,v) \mid u \in V \right\}$$

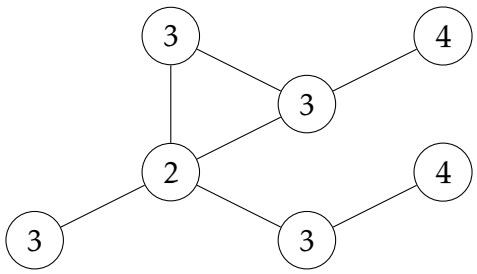


Figure 4

In Figure 4, each vertices' eccentricity is written on their nodes. The radius of the graph is 2, which is the minimum eccentricity, and the diameter of the graph is 4, which is the maximum eccentricity. The eccentricity sequence of the graph is $(2,3,3,3,3,4,4)$. It can also be expressed as $(2^1, 3^4, 4^2)$.

3 Types of Graphs

Complete Graph

A complete graph is a graph in which all vertices are connected to each other. Its radius is 1, and its diameter is also 1. It's eccentricity sequence is $(1,1,\cdots,1)$, which has $|V|$ 1s.

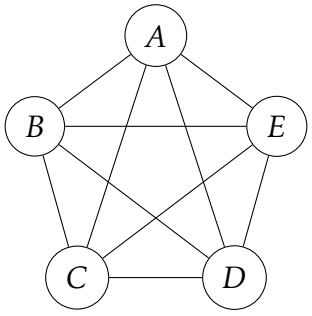


Figure 5

Cycle Graph

A cycle graph contains a single cycle through all vertices. Its radius is $\lfloor \frac{|V|}{2} \rfloor$ and diameter is also $\lfloor \frac{|V|}{2} \rfloor$. Its eccentricity sequence is $(\lfloor \frac{|V|}{2} \rfloor, \lfloor \frac{|V|}{2} \rfloor, \cdots, \lfloor \frac{|V|}{2} \rfloor)$, which has n $\lfloor \frac{|V|}{2} \rfloor$s.

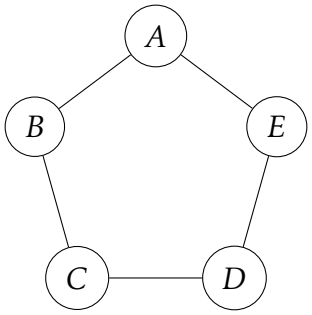


Figure 6

Path Graph

A path graph is a graph whose vertices can be listed in the order $v_1, v_2, \cdots, v_{|V|}$ such that the edges are $\{v_i, v_i + 1\}$ where $i = 1, 2, \cdots, |V| - 1$. Its radius is $\lfloor \frac{|V|}{2} \rfloor$ and diameter is $|V| - 1$. Its eccentricity sequence is $(r, r, r+1, r+1, \cdots, d)$, where r is its radius and d is its diameter. d appears one or two times in the eccentricity sequence.

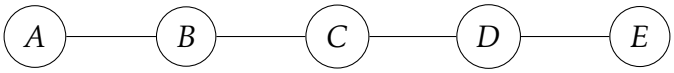


Figure 7

Tree

A tree is a connected graph with no cycles.

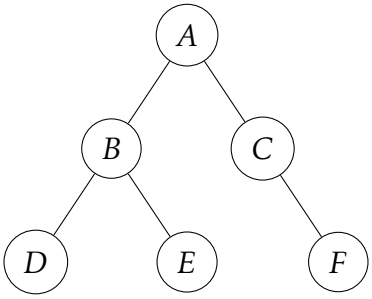


Figure 8: tree example

In Figure 8, the parent node of B is A, and the child nodes of B are D and E. If some node has no parent nodes, it is called the root node. The root node should be

uniquely determined. Nodes with no child nodes are called leaf nodes. Thus, the root node is A, and the leaf nodes are D, E, and F.

Lemma 10. *In a tree, for every pair of distinct vertices (u,v), there exists a unique path having u and v as the initial and terminal vertices.*

Proof. Assume that there exists two different paths $(u,p_1,p_2,\cdots,p_x,v)$ and $(u,q_1,q_2,\cdots,q_y,v)$. Since we can make a cycle $(u,p_1,p_2,\cdots,p_x,v,q_y,q_{y-1},\cdots,q_1)$ by connecting the second path reversely, it contradicts with the definition of trees.

Lemma 11. *For any tree $G=(V,E)$, the following holds. $|E|=|V|-1$*

Proof. Let $G=(V,E)$ be a tree such that $|V|=n$. If $n=1$, since it is only a single vertex, $|E|=n-1$. If $n=2$, G is a graph with two vertices and one edge connecting each other. Therefore, $|E|=n-1$. The statement is true for $n=1,2$.

Assume the statement is true when $n=1,2,\cdots,k$. Let e be an edge connecting two vertices, v and u. v and u have only one path by the Lemma. Therefore, if e is deleted, G would be disconnected into two components $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$. Since G is divided to G_1 and G_2, $|V_1|+|V_2|=k+1$ where $|V_1|<k+1$ and $|V_2|<k+1$. The number of total edges $|E|=|E_1|+|E_2|+1=(|V_1|-1)+(|V_2|-1)+1$ (by assumption) $=k-2+1=k-1$. The statement is also true when $n=k+1$.

By complete induction, the statement is true for all trees.

Cactus

A cactus is a connected graph such that no common edges are shared by any two different cycles. However, there may be common vertices for some cycles.

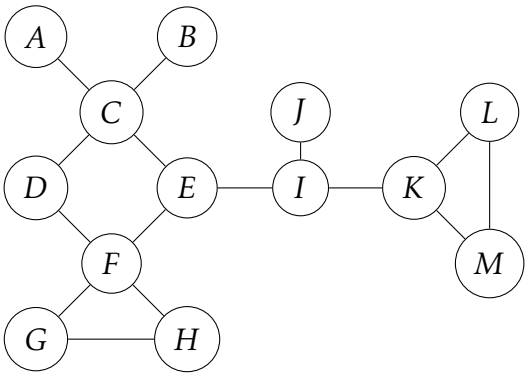


Figure 9

4 Background Algorithms

Breadth-First Search

Breadth-First Search(BFS) is an algorithm for the traversal of graphs. By using a modified BFS process, we can compute the distances efficiently.

input : $G = (V, E)$, $s \in V$
output: D

```
1  D ← [NULL] × |V|;
2  Q ← empty queue;
3  D[s] ← 0;
4  insert s to Q;
5  while Q is not empty do
6      u ← front element of Q;
7      remove front element of Q;
8      foreach v is adjacent to u do
9          if D[v] ≠ −1 then
10             continue;
11         end
12         D[v] ← D[u] + 1;
13         insert v to Q;
14     end
15 end
16 return D;
```

Algorithm 1: GetDistance

Algorithm 1 gets a graph and a vertex s as input and outputs a list D satisfying $D[x] = d(s,x)$. Since the 8th line requires $O(|E|)$ time to scan all edges, the algorithm should take an adjacency list of the graph as input. For example, the adjacency list of Figure 10 is $[A : [B,C,D], B : [A,E], C : [A,F], D : [A,H], E : [B,F], F : [C,E,G], G : [F], H : [D]]$. Using a FIFO(First In, First Out) data structure like a queue, the vertices are traversed in the order of ascending distances from the closest vertex to the farthest vertex. In Figure 10, the vertices are inserted into the queue as follows.

- A is inserted (the initial vertex; distance is 0)
- B, C, and D are inserted (distance is 1)
- E, F, and H are inserted (distance is 2)
- G is inserted (distance is 3)

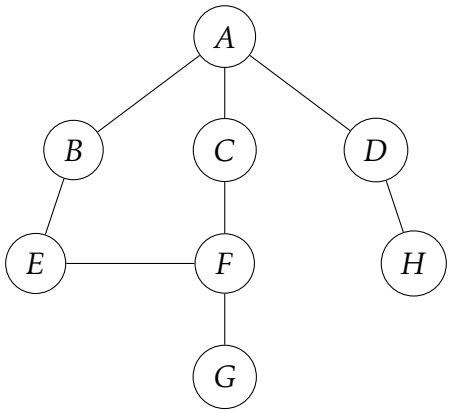


Figure 10

After the BFS process, the list D contains the distances having s as the initial vertex. If some value $D[x]$ is -1, the vertex x is unreachable from s; x and s are not in the same connected component.

The time complexity of the algorithm is linear. Since each vertex is traversed a maximum of once, $O(|V|)$ time is required. From Lemma 2(Handshaking Lemma), the total sum of the degrees is proportional to $|E|$, thus adding $O(|E|)$ to the time complexity. The overall time complexity is $O(|V|+|E|)$.

Diameter of Trees

To find out a diameter of a tree, it is possible to run $|V|$ BFS processes, but it requires $O(|V|) \times O(|V|+|E|) = O(|V|^2+|V||E|)$ time, which is inefficient. Therefore, we use the following proposition.

Lemma 12. *In a tree, the farthest vertex from any arbitrary vertex is always one of a diameter's two endpoints.*

Proof. Let G be a tree, and a and b be end vertices of a diameter d. For a random vertex v, let u be the farthest vertex from v. Then, u is a leaf node.

Suppose no common vertex is included in the path between a and b and the path between v and u. Let s be a vertex in the path between a and b, and t be a vertex included in the path between v and u, such that there is no common edge included in the path between a and b, s and t, and v and u. Then, $d(v,t)+d(t,u) = d(v,u) \geq d(v,b) = d(v,t)+d(t,s)+d(s,b)$ and $d(t,u) \geq d(t,s)+d(s,b)$. Also, $d(a,s)+d(s,b) = d(a,b) \geq d(a,u) = d(a,s)+d(s,u)$ and $d(s,b) \geq d(s,u)$. Therefore, $d(t,u) \geq d(t,s)+d(s,b) \geq d(t,s)+d(s,u)$, which is a contradiction.

Based on the above, a common vertex w is included in the path between a and b and the path between v and u. Then, $d(v,w)+d(w,u)=d(v,u)\geq d(v,b)=d(v,w)+d(w,b)$. Therefore, $d(w,u)\geq d(w,a)$, and $d(a,b)=d(a,w)+d(w,b)\leq d(a,w)+d(w,u)=d(a,u)$. Since $d(a,b)$ is a diameter, $d(a,b)\geq d(a,u)$. Hence, $d(a,b)=d(a,u)$ and u is an endpoint of a diameter of G.

input : $G=(V,E)$
output: (v_1,v_2)

```
1  u ← random vertex;
2  D ← GetDistance(G, u);
3  m ← −1;
4  foreach v ∈ V do
5      if m ≤ D[v] then
6          m ← D[v];
7          v1 ← v;
8      end
9  end
10 D′ ← GetDistance(G, v1);
11 m ← −1;
12 foreach v ∈ V do
13     if m ≤ D′[v] then
14         m ← D′[v];
15         v2 ← v;
16     end
17 end
18 return (v1, v2);
```

Algorithm 2: GetDiameter

Algorithm 2 above gets a graph as input, and outputs a pair of vertices (v_1,v_2) where v_1 and v_2 are two endpoints of the diameter. The process requires only two calls of `GetDistance()` because of Lemma 12. Since the time complexity of both for-each loops in lines $4-9$ and $12-17$ are less than or equal to $O(|E|)$, the overall time complexity is $O(|V|+|E|)$.

Detecting Cycles in a Cactus

We can find out cycles in a cactus by Depth-First Search(DFS). If we visit the same vertex twice or more, it means there exists a cycle that contains the vertex. Since we have searched the vertices by depth, we can simply find vertices included in the cycle by tracking backward with its parent vertices.

input : $G = (V, E)$
output: C

```
1  s ← random vertex;
2  S ← empty stack;
3  P, ord ← [NULL] × |V|, [NULL] × |V|;
4  M ← empty queue;
5  C ← [];
6  i ← 0;
7  c ← 0;
8  insert s to S;
9  while S is not empty do
10     u ← front element of S;
11     remove front element of S;
12     ord[u] ← c;
13     c ← c + 1;
14     foreach v is adjacent to u do
15         if v is P[u] then
16             continue;
17         end
18         if v has already visited then
19             if ord[v] < ord[u] then
20                 insert (v, u) to M;
21             end
22         end
23         else
24             P[v] ← u;
25             insert v to S;
26         end
27     end
28 end
29 while M is not empty do
30     insert [] to C;
31     (v_init, v_term) ← front element of M;
32     remove front element of M;
33     v ← v_term;
34     while v is not v_init do
35         insert v to C[i];
36         v ← P[v];
37     end
38     insert v_init to C[i];
39     i ← i + 1;
40 end
41 return C;
```

Algorithm 3: DetectingCycles

Algorithm 3 gets a graph as input, and outputs a list of cycles. The cycles can be expressed as several vertices $(v_1, v_2, ..., v_n)$ contained by them. We can detect cycles in a cactus in two steps. In the first step, we do DFS and save parent of each vertices. If we visit a same vertex v_{init} twice, we memo the pair (v_{init}, v_{term}) where v_{term} is the parent of v_{init}. In the second step, we can get cycles with the pairs which we memoed before. For a pair, we can track parent vertices one by one, starting from v_{term}, while we get v_{init}. Then, we can get vertices contained by a cycle.

Sliding Window

"Given a number sequence $(a_1, a_2, \cdots, a_n)$ *and an integer* $1 \leq \ell \leq n$*, calculate* $X_i = \max_{k=i}^{i+\ell-1} a_i$ *for* $i = 1, 2, \cdots, n-\ell+1$*. In other words, compute the maximum of all subarrays of length* ℓ*."*

A naïve algorithm for the problem is calculating the asked value in $O(n)$ time complexity for each value of i. When $\ell \approx \frac{n}{2}$, the calculation requires approximately $\frac{n^2}{4}$ operations, which leads to the time complexity of $O(n^2)$.

An efficient algorithm uses Sliding Window. The numbers in $(a_1, a_2, \cdots, a_\ell)$ and $(a_2, a_3, \cdots, a_{\ell+1})$ only differ by a_1 and $a_{\ell+1}$, which means we can save time by using the previous result for X_{i-1} to calculate X_i.

input : $(a_1, a_2, \cdots, a_n), 1 \leq \ell \leq n$
output: X

```
1  Dq ← empty deque;
2  X ← [0] × n;
3  for i ← 1 to n do
4      while Dq is not empty ∧ front element of Dq ≤ i − ℓ do
5          remove front element of Dq;
6      end
7      while Dq is not empty ∧ a_{back element of Dq} < a_i do
8          remove back element of Dq;
9      end
10     insert i to back of Dq;
11     if i ≥ ℓ then
12         X[i − ℓ + 1] ← a_{front element of Dq};
13     end
14 end
15 return X;
```

Algorithm 4: SubarrayMaximum

Algorithm 4 illustrates Sliding Window. The algorithm gets the sequence $(a_1, a_2, \cdots, a_n)$ and integer ℓ as input, and outputs $X = [X_1, X_2, \cdots, X_{n-\ell+1}]$. The double-ended queue(deque) Dq contains the indices of a_i that could be the maximum value sometime in the future(i.e., the candidates). Also, the front element of Dq should be the answer. The

indices that cannot reach the maximum should be removed to achieve this. The elements are in ascending order since the insertion only occurs in the back of Dq. Therefore, by popping out the front element of Dq while its value is less than or equal to $i-\ell$, we can leave only the valid ones, the indices between $i-\ell+1$ and i(inclusive). Let x be the back element of Dq. If a_x is smaller than a_i, a_x can never reach the maximum, thus a_x should be removed.

Repeating the process for $i = 1, 2, \cdots, n$ and getting answer from the front element of Dq when $i \geq \ell$, all of $X_1, X_2, \cdots, X_{n-\ell+1}$ are calculated in $O(n)$ time complexity. This is because there are a maximum of only $2n$ insertions and deletions.

Counting Sort

Counting Sort is an algorithm that quickly sorts the elements of A in non-decreasing order, assuming that there is a set of numbers in A.

The process of output coming out when input comes into the function is as follows. $A = \{1, 2, 4, 5, 1, 3, 2, 2, 6\}$ The number of each elements is $1:2, 2:3, 3:1, 4:1, 5:2, 6:1$. If you arrange them in the count order, sorted list B becomes $\{1, 1, 2, 2, 2, 3, 4, 5, 6\}$.

input : A
output: B

```
1  B ← [];
2  n ← the mininum element of A;
3  m ← the maximum element of A;
4  C ← [0] × (m − n + 1);
5  for a ← element of A do
6      C[a − n] ← C[a − n] + 1;
7  end
8  for i ← 0 to m − n do
9      while C[i] > 0 do
10         insert n + i to B;
11         C[i] ← C[i] − 1;
12     end
13 end
14 return B;
```

Algorithm 5: CountingSort

Algorithm 5 is a pseudo code of Counting Sort. It assumes that the elements of A are only integers. Depending on the condition of the elements, the complexity of the code varies.

5 Eccentricity Sequence of Trees

First, a diameter of a tree can be found using Algorithm 2. The figure below shows the distance to reach each point starting from the initial vertex of a diameter.

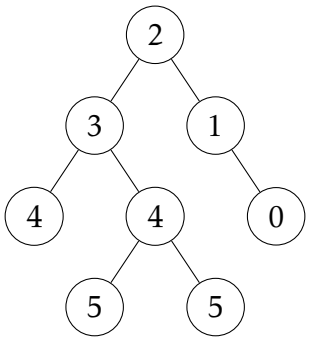


Figure 11

Next, the distances starting from the terminal vertex of the diameter is obtained again using BFS. The distance to each vertex is marked below.

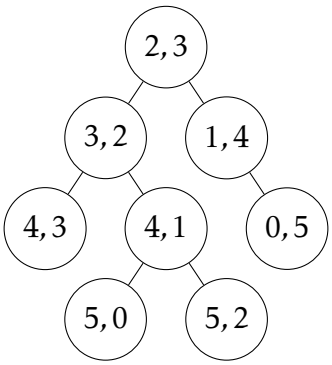


Figure 12

Finally, compare the two numbers shown at each vertex and leave only the larger numbers. The largest of these numbers at each vertex is the eccentricity to be sought.

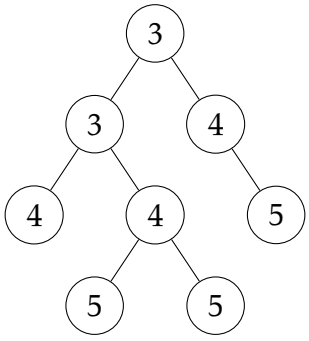


Figure 13

6 Eccentricity Sequence of Cacti

Let $\Gamma = \{\Gamma_1, \Gamma_2, \cdots, \Gamma_{|\Gamma|}\}$ be the set of cycles in a cactus.

6.1 $|\Gamma| = 1$

If a cactus has only one cycle, its shape is a graph made by adding an edge to a tree. This is true because removing an edge contained in the cycle results in a connected graph with no cycle; a tree. Therefore, from Lemma 11, the number of edges is $(|V| - 1) + 1 = |V|$ when $|\Gamma| = 1$.

An example for $|\Gamma| = 1$ is shown below.

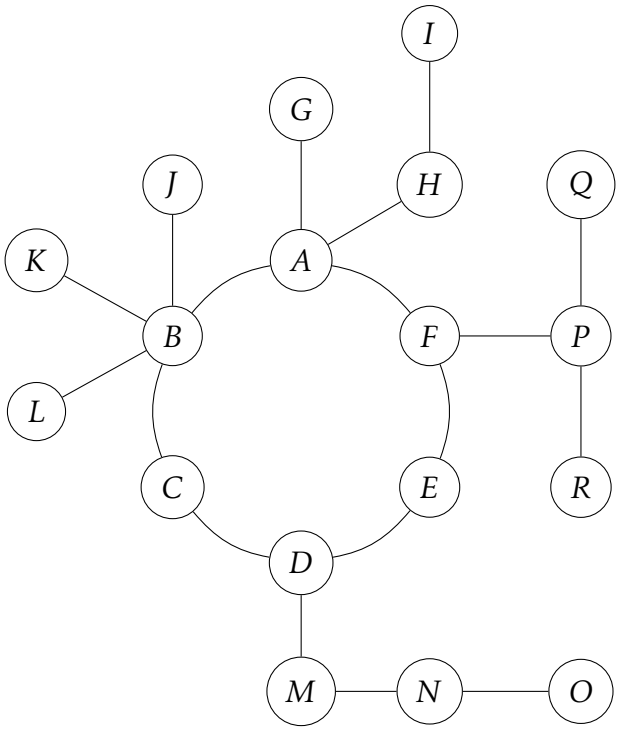


Figure 14

The cycle Γ_1 is (A, B, C, D, E, F) and there are no other cycles in the graph. If we remove 6 edges contained in the cycle, the graph is no longer connected and it is separated to 6 disjoint trees. Thus, the general shape of a cactus with $|\Gamma| = 1$ is represented as Figure 15.

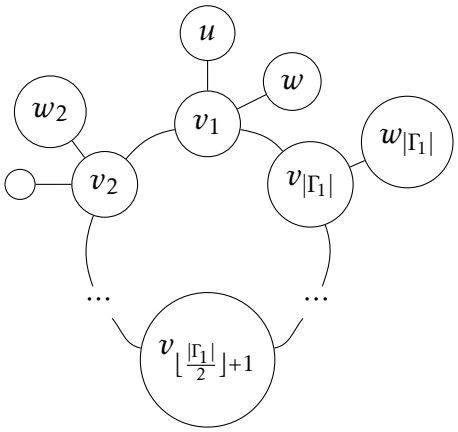


Figure 15

There is a single cycle with length $|\Gamma_1|$. Also, for each vertex v_i in the cycle, a tree is attatched to v_i(i.e. the root of the tree is v_i). We will call the tree with root v_i as T_i. Although the figure shows 0, 1, or 2 edges connected to v_i(excluding the cycle's edges), there could be more than 2 edges. For the vertex $u \in T_1$, $\lfloor \frac{|\Gamma_1|}{2} \rfloor + 1$ candidates exist for the longest path when we assume that the cycle should be traversed only counterclockwise. (Let $h = \lfloor \frac{|\Gamma_1|}{2} \rfloor$ for convenience.)

#0. $(u, \cdots, w)$ — the longest path inside the tree T_1

#1. $(u, \cdots, v_1, v_2, \cdots, w_2)$ — the path passing through v_1 and v_2 where w_2 is the deepest vertex in T_2

#2. $(u, \cdots, v_1, v_2, v_3, \cdots, w_3)$ — the path passing through v_1, v_2, and v_3 where w_3 is the deepest vertex in T_3
...

#h. $(u, \cdots, v_1, v_2, v_3, \cdots, v_{h+1}, \cdots, w_{h+1})$ — the path passing through $v_1, v_2, \cdots, v_{h+1}$ where w_{h+1} is the deepest vertex in T_{h+1}

Using the eccentricity algorithm for trees in T_1, the length of path #0 can be calculated in $O(1)$ using precomputation. Let the depth of some vertex x as d_x. Then, the maximum depth in tree T_i is $\max_{v \in T_i} d_v$. Let this value M_i. The length of path #1, #2, #3, $\cdots$, #h are $d_u + 1 + M_2, d_u + 2 + M_3, d_u + 3 + M_4, \cdots, d_u + h + M_{h+1}$, respectively. Thus, we need to find out the value of $\max_{k=2}^{h+1} (d_u + (k-1) + M_k)$ efficiently. When $u \in T_2$, the expression changes to $\max_{k=3}^{h+2} (d_u + (k-2) + M_k)$. And when $u \in T_i$, the expression changes to $\max_{k=i+1}^{h+i} (d_u + (k-i) + M_k)$, where M_x with $x > |\Gamma_1|$ is considered as $M_{x-|\Gamma_1|}$. Let $f(i)$ be $\max_{k=i+1}^{h+i} (k-i) + M_k$. If we replace $k + M_k$ by M'_k, $f(i) = \max_{k=i+1}^{h+i} \left(M'_k - i\right)$. Since $-i$ does not vary when k moves between $i+1$ and $h+i$, it is possible to write $f(i) = -i + \max_{k=i+1}^{h+i} M'_k$. Looking closer at $\max_{k=i+1}^{h+i} M'_k$, it is the maximum in a subarray of M' with length h. From Sliding Window (Algorithm 4), we can calculate all of $f(1), f(2), \cdots, f(|\Gamma_1|)$ in $O(|\Gamma_1|)$ time complexity. [1] After the

[1] It is also possible to use a modified algorithm of Sliding Window to handle the circular shape, but it is much more simple to concatenate the array with a copy of itself. This technique is used for numerous problems involving circles.

calculation, traversing the whole graph and adding d_u for each vertex u works.

Since we only considered counterclockwise traverses on the cycle, by reversing $d_1, d_2, \cdots, d_{|\Gamma_1}$ and repeating the same process gives the effect of clockwise traverses. The time required for each step is the following:

- $\sum O(|T_i|) = O(|V|)$ — for the precomputation of eccentricites in trees
- $O(|\Gamma_1|)$ — Sliding Window
- $O(|V|)$ — the final process for adding depth to each vertex's result

Therefore, the overall time complexity is $O(|V|)$. As mentioned in the eccentricity sequence algorithm for trees, the final sorting can be performed in $O(|V|)$ time using Counting Sort.

6.2 $|\Gamma| = 2$

A cactus with two cycles is shaped like a dumbbell, because the whole graph is consisting of two cycles and a single tree connecting the cycles. The general representation of the cactus with $|\Gamma| = 2$ is the figure below.

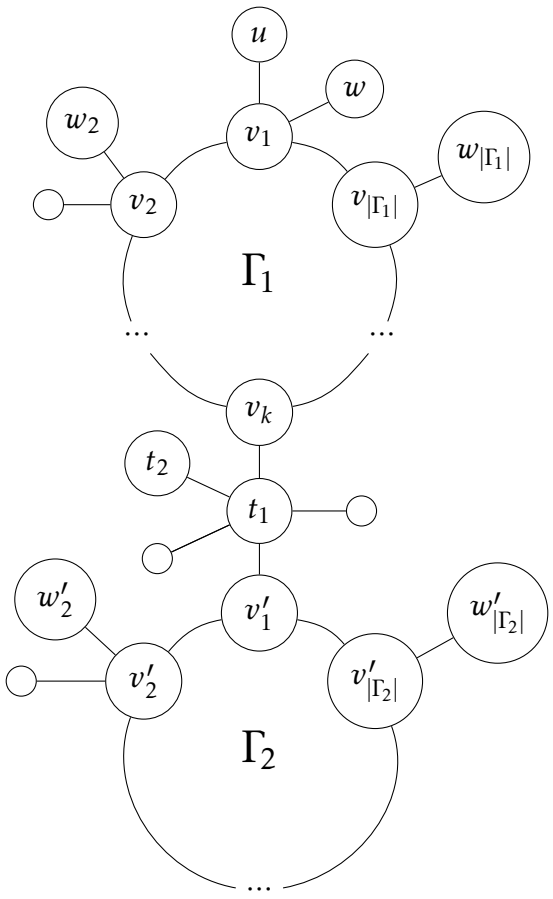


Figure 16

In the graph obtainted by removing all the edges belonging to $\Gamma \setminus \Gamma_1$, let $\Omega(\Gamma_1)$ be the connected component containing Γ_1. Similarly define $\Omega(\Gamma_2)$. Let's take the part $\Omega(\Gamma_1)$ of the graph, which contains the cycle Γ_1, the trees attached to Γ_1. Then, $\Omega(\Gamma_1)$ satisfies $|\Gamma| = 1$, and it is the same situation explained in Case 1. Therefore, we can get the eccentricity sequence of $\Omega(\Gamma_1)$ in linear time. Let v_1' be a vertex included in both $\Omega(\Gamma_1)$ and Γ_2. From the perspective of cycle Γ_2, $\Omega(\Gamma_1)$ can be considered as a tree. Now, we can replace $\Omega(\Gamma_1)$ with a tree in which the maximum depth is equal to the eccentricity of v_1'. After the replacement, simply perform the method we found for the case $|\Gamma| = 1$. This process calculates the eccentricities for the vertices in $\Omega(\Gamma_2) \setminus \Omega(\Gamma_1)$. But since the eccentricities in $\Omega(\Gamma_1)$ could be modified because of the new cycle Γ_2, run BFS to update all of the eccentricities of $\Omega(\Gamma_1)$.

We can generalize this as three steps. First, find the eccentricity for the vertices in $\Omega(\Gamma_1)$. Second, add $\Omega(\Gamma_2)$ on $\Omega(\Gamma_1)$ and find the eccentricity of the vertices in $\Omega(\Gamma_2) \setminus \Omega(\Gamma_1)$. Third, update the eccentricity of the vertices in $\Omega(\Gamma_1)$.

The time complexity required in each step is shown below.

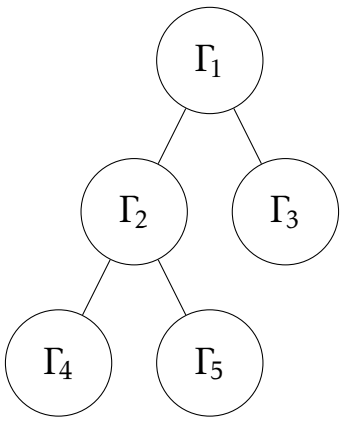
- $O(|\Omega(\Gamma_1)|)$ — the computation of eccentricities of $\Omega(\Gamma_1)$
- $O(|\Gamma_2|)$ — Sliding Window
- $O(|\Omega(\Gamma_1)| + |\Gamma_2|) = O(|V|)$ — the computation of eccentricities of $\Omega(\Gamma_2) \setminus \Omega(\Gamma_1)$ and the last BFS process for updating $\Omega(\Gamma_1)$

The algorithm still uses linear time($O(|V|)$), since there are only two cycles.

6.3 $|\Gamma| > 2$

If there are more than two cycles(i.e., $|\Gamma| > 2$), we can think of the whole graph as a tree of cycles. Since every edge belongs to a maximum of one cycle, the cycle sequence passed through by the path connecting any two cycles is determined uniquely. Thus, it is possible to convert the cactus into a tree having cycles as vertices.

By representing cycles as circles, the general shape of a cactus is the following.



The situation of $|\Gamma| > 2$ is merely an extension of the situation of $|\Gamma| = 2$. First, obtain the parts: $\Omega(\Gamma_1), \Omega(\Gamma_2), \cdots, \Omega(\Gamma_{|\Gamma|})$. Second, attach each part one by one. The new part must be connected with the original part to replace the original part with a tree. As we attach each part, update the eccentricity of the vertices with the algorithms explained above. From the paragraph above, it is guaranteed that adding a new adjacent part would not make infinite loops. Finally, use Counting Sort to get the eccentricity sequence of the whole graph.

7 Conclusion

In conclusion, with the idea of representing the cactus as a tree of cycles, and the method of finding eccentricities centered on a cycle, we can obtain the eccentricity sequence of cacti in $O(|\Gamma||V|)$ time complexity. Since $|\Gamma| < \frac{|V|}{2}$ holds, the method we found is more efficient than running BFS for $|V|$ times, which requires $O(|V|^2)$ time. The C++/Python implementation of the method and report of our research is available at https://github.com/modularinv/eccseq-algo/.

8 Reference

1. https://www.geeksforgeeks.org/some-theorems-on-trees/
2. https://mathworld.wolfram.com/CactusGraph.html