Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 19, no. 1, pp. 11–41 (2015)
DOI: 10.7155/jgaa.00345

Optimal Data Structures for Farthest-Point Queries in Cactus Networks

Prosenjit Bose[1] *Jean-Lou De Carufel*[1] *Carsten Grimm*[1,2]
Anil Maheshwari[1] *Michiel Smid*[1]

[1]Carleton University, Ottawa, Ontario, Canada
[2]Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany

Abstract

Consider the continuum of points on the edges of a network, i.e., a connected, undirected graph with positive edge weights. We measure the distance between these points in terms of the weighted shortest path distance, called the *network distance*. Within this metric space, we study farthest points and farthest distances. We introduce optimal data structures supporting queries for the farthest distance and the farthest points on trees, cycles, uni-cyclic networks, and cactus networks. Using only linear space and construction time, we support farthest-point queries in $\Theta(k)$ time for trees, in $\Theta(\log n)$ time for cycles, and in $\Theta(k + \log n)$ time for uni-cyclic networks and cactus networks, where n is the size of the network and k is the number of farthest-points.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
August 2014	November 2014	December 2014	December 2014	January 2015

Published:
January 2015

Article type:	Communicated by:
Regular paper	S. Kobourov

This research has been partially funded by NSERC, FQRNT, and DAAD. A preliminary version of this work was presented at the 25th Canadian Conference on Computational Geometry [4] and was part of the Diplomarbeit (Master's thesis) of the third author [9].

E-mail addresses: jit@scs.carleton.ca (Prosenjit Bose) jdecaruf@cg.scs.carleton.ca (Jean-Lou De Carufel) carsten.grimm@ovgu.de (Carsten Grimm) anil@scs.carleton.ca (Anil Maheshwari) michiel@scs.carleton.ca (Michiel Smid)

1 Introduction

Consider the continuum of points on the edges of a network, i.e., a graph with positive edge weights. We measure the distance between these points in terms of the weighted shortest path distance, called the *network distance*. Within this metric space, we study farthest points and farthest distances.

Decisions where to place facilities are often complex and involve optimization of multiple criteria. Our data structures enable decision makers to quickly compare farthest distances from potential locations, which may constitute an essential factor, e.g., impacting emergency response times for a possible location of a new hospital. Furthermore, our results provide a *heat-map* of farthest distances illuminating the aspect of centrality of the network at hand and, thereby, serve as a visual aid for decision makers.

We introduce optimal data structures supporting queries for the farthest distance and the farthest points on trees, cycles, uni-cyclic networks, and cactus networks. We begin with data structures for simple networks and then use them as building blocks for more complex networks. With this modular approach we can easily extend our results as new building blocks become available.

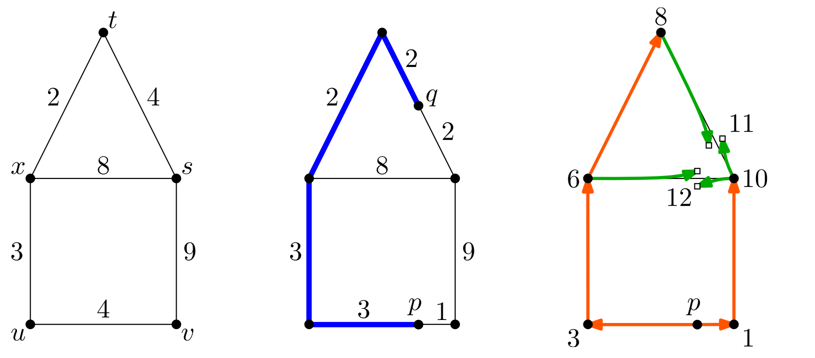
The remainder of this section is organized as follows. In Section 1.1, we set the stage for this work by making our notions of networks, points along networks, and network distance precise. In Section 1.2, we summarize related work. In Section 1.3, we outline our main contributions and the structure of this work.

1.1 Preliminaries and Problem Definition

We call a simple, finite, undirected graph with positive edge weights a *network*. Unless stated otherwise, we consider only connected networks. Let $G = (V, E)$ be a network with n vertices and m edges, where V is the set of vertices and E is the set of edges. We write uv to denote an edge with endpoints $u, v \in V$ and we write w_{uv} to denote its weight. A point p on edge uv subdivides uv into two sub-edges up and pv with $w_{up} = \lambda w_{uv}$ and $w_{pv} = (1 - \lambda)w_{uv}$, where λ is the real number in $[0, 1]$ for which $p = \lambda u + (1 - \lambda)v$. We write $p \in uv$ when p is on edge uv and $p \in G$ when p is on some edge of G.

As illustrated in Figure 1, we measure distance between points $p, q \in G$ as the weighted length of a shortest path from p to q in G, denoted by $d_G(p, q)$. We say that p and q have *network distance* $d_G(p, q)$. The points on G and the network distance form a metric space. Within this metric space, we study farthest points and farthest distances. We call the largest network distance from some point p on G the *eccentricity* of p and denote it by $\mathrm{ecc}_G(p)$, i.e., $\mathrm{ecc}_G(p) = \max_{q \in G} d_G(p, q)$. A point $\bar{p}$ on G is farthest from p if and only if $d_G(p, \bar{p}) = \mathrm{ecc}_G(p)$. We omit the subscript G whenever the network is understood from the context.

We extend the definition of shortest path trees from graphs to networks. When we say we *cut* an edge ab at a point $p \in ab$ we mean that we introduce two new vertices x and y and replace the edge ab with two edges ax and yb of weight $w_{ax} = w_{ap}$ and $w_{yb} = w_{pb}$. If p coincides with an endpoint of ab, then one of ax and yb has weight zero and is omitted. Let s be some point on a network



(a) A network. (b) A shortest path from p to q. (c) A shortest path tree with its extension.

Figure 1: In the network shown in (a), the network distance from $p = \frac{1}{4}u + \frac{3}{4}v$ to $q = \frac{1}{2}s + \frac{1}{2}t$ is $d(p, q) = 10$. This distance is achieved along the shortest path (blue) from p to q depicted in (b). The eccentricity of p is $\text{ecc}(p) = 12$ and the farthest point from p lies along the edge xs, as indicated by the shortest path tree from p (orange) and its extension (orange $\cup$ green) illustrated in (c).

G, and let T_s be the shortest path tree of s in G. We split each non-T_s edge ab at the farthest point from s on ab and add the resulting edges ax and yb to T_s. The resulting tree is called the *extended shortest path tree* [26] of s in G; this tree encapsulates both the eccentricity of s in G and the farthest points from s in G, as illustrated in Figure 1c.

We aim to construct data structures for a fixed network G supporting the following queries. Given a point p on G, what is the eccentricity of p? What is the set of farthest points from p in G? We refer to the former as an *eccentricity query* and to the latter as a *farthest-point query.* Both queries consist of the query point p represented by the edge uv containing p and the value $\lambda \in [0, 1]$ such that $p = \lambda u + (1 - \lambda)v$. We report farthest points in the same format, i.e., for each farthest point $\bar{p}$ from p, we state an edge xy containing $\bar{p}$ and a value $\mu \in [0, 1]$ specifying the exact position of $\bar{p}$ along xy.

We study trees, cycles, uni-cyclic networks, and cactus networks. A *uni-cyclic network* is a network with exactly one simple cycle. A *cactus network* is a network in which no two simple cycles share an edge or, in other words, a network where each edge is contained in at most one simple cycle.

1.2 Related Work

Our data structures implicitly represent (generalized) farthest-point network Voronoi diagrams, where the sites are the entire continuum of points along a network [3]. Usually, Voronoi diagrams are defined with respect to a finite set of sites representing points of interest in some metric space [27, 1]. The existing research on Voronoi diagrams on networks covers a wide range of queries including queries for the closest [12, 28], the farthest [26, 7], and the k-th nearest neighbors [21, 8, 31] among a finite set of sites. We refer the reader to Okabe et al. [27], Taniar and Rahayu [31], Okabe and Suzuki [25], and Okabe and Sugihara [24] for guides through the literature on network Voronoi diagrams.

Network Voronoi diagrams relate to center problems from location analysis [12, 25]. Let $G = (V, E)$ be a network. A *center* [11] of G is a vertex v that minimizes the network distance to any other vertex, i.e., $\max_{u \in V} d(u, v) = \min_{x \in V} \max_{u \in V} d(u, x)$. An *absolute center* a generalizes a center in that it may be placed anywhere along edges of the network, i.e., $\max_{u \in V} d(u, a) = \min_{p \in G} \max_{u \in V} d(u, p)$. A *continuous absolute center* is a point c with minimal eccentricity, i.e., $\max_{p \in G} d(p, c) = \min_{q \in G} \max_{p \in G} d(p, q) = \min_{q \in G} \mathrm{ecc}(q)$.

Our results draw three ingredients from the literature on center problems: First, the absolute center [14] plays a crucial role when querying for farthest points in a tree network, as we shall see in Section 2.1. Second, the decomposition of cactus networks into *blocks*, *branches*, and *hinges* [17, 23, 5, 2] helps us exploit their tree structure [16, 19, 10] in Section 3. Third, viewing the network from different *perspectives* [13, 20, 23, 2] allows us to process branches and blocks independently, as explained in Sections 2.3 and 3.

Conversely, our constructions solve some center problems *en passant*: We obtain the absolute center of a uni-cyclic network [13] as a by-product when building our data structure for queries in uni-cyclic networks. While it was known how to locate a single continuous absolute center in a cactus network in linear time [2], we produce the entire set of these centers in $O(n)$ time improving the $O(m^2 \log n)$ bound for general networks with n vertices and m edges [15].

A comprehensive summary of the literature about center problems is beyond the scope of this work. We refer readers interested in this sub-field of location analysis to a wealth of surveys [29, 33, 34, 18, 22], as well as to treatments of more recent results in the books by Kincaid [19], Tansel [32], and Shi [30].

1.3 Structure and Results

We introduce data structures supporting eccentricity queries and farthest-points queries for trees, cycles, uni-cyclic networks, and cactus networks. Table 1 summarizes the characteristics of these data structures and compares them to our previous result for general networks [3]. The data structures presented in this work have optimal size and construction times, as well as optimal query times. The lower bounds of $\Omega(\log n)$ stem from reducing predecessor search to locating the sub-edge containing a query point along a subdivided edge.

The remainder of this work is organized as follows. In Section 2, we introduce data structures for trees, cycles, and uni-cyclic networks. In Section 3, we construct data structures supporting eccentricity queries and farthest-point queries on cactus networks. Our approach is to reduce a cactus network to smaller networks having a sufficiently simple structure such that the data structures and query algorithms of Section 2 can be applied. In Section 4, we discuss directions for future research on closing the gap between general networks and cactus networks in Table 1.

Network	Eccentricity	Farthest-Point	Size	Construction
Tree	$\Theta(1)$	$\Theta(k)$	$\Theta(n)$	$\Theta(n)$
Cycle	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Uni-Cyclic	$\Theta(\log n)$	$\Theta(k + \log n)$	$\Theta(n)$	$\Theta(n)$
Cactus	$\Theta(\log n)$	$\Theta(k + \log n)$	$\Theta(n)$	$\Theta(n)$
General	$\Theta(\log n)$	$\Theta(k + \log n)$	$O(m^2)$	$O(m^2 \log n)$

Table 1: The traits of the data structures for queries in different types of networks, where n is the number of vertices, m is the number of edges, and k is the number of farthest-points. The data structures for trees, cycles, uni-cyclic networks, and cactus networks are the matter of this work, whereas the data structure for general networks was introduced in previous work [3]

2 Trees, Cycles, and Uni-Cyclic Networks

In this section, we introduce data structures for farthest-point queries in trees and cycles. We then combine these two structures to support queries in uni-cyclic networks, i.e., networks with exactly one simple cycle. These data structures have linear size and construction times while providing optimal query times, and they serve as building blocks for our data structure for cactus networks.

2.1 Trees

The layout of farthest points on a tree hinges on the position of the absolute center. This point subdivides a tree into sub-trees where all points in a given sub-tree have their farthest points in other sub-trees. Conversely, each sub-tree has a set of leaves that are farthest from the absolute center and these leaves will be farthest from points in other sub-trees.

On tree networks, every farthest point is a leaf and the point c whose farthest leaves are closest is an *absolute center* [11, 20]. In other words, an absolute center on a tree T is a point with minimal eccentricity, i.e., $\text{ecc}(c) = \min_{g \in T} \text{ecc}(g)$. We say that two leaves l and l' of T are *most distant* when they realize the maximum distance between two points on T, i.e., $d(l, l') = \max_{a,b \in T} d(a, b)$.

Theorem 1 (Handler [14]) *Every tree has exactly one absolute center midway along any path connecting two most distant leaves and we can locate the absolute center of a tree with n vertices in $O(n)$ time.*

Handler [14] determines the absolute center of a tree with two rounds of breadth-first-search: The first breadth-first-search starts from an arbitrary leaf l. With this search, we determine a farthest leaf $\bar{l}$ from l. We then start the second breath-first-search from $\bar{l}$ to determine a farthest leaf $\hat{l}$ from $\bar{l}$. Handler [14] shows that $\bar{l}$ and $\hat{l}$ are most distant leaves and that the absolute center is located midway on the path from $\bar{l}$ to $\hat{l}$.

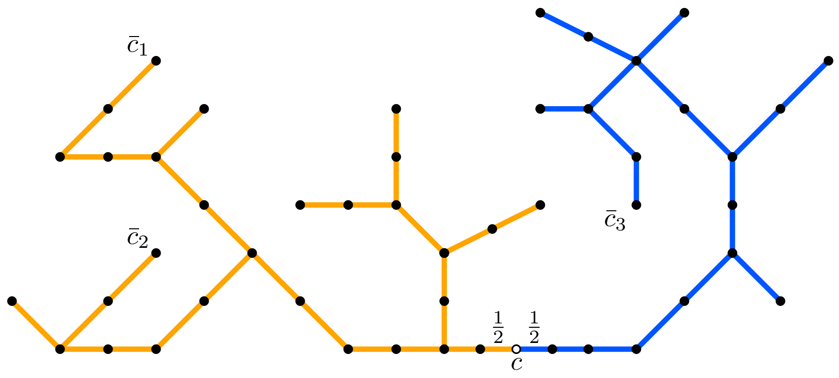


Figure 2: A tree T where all edges have unit weight unless indicated otherwise. The absolute center c splits T into two sub-trees T_1 (orange) and T_2 (blue). The farthest distance from c is $\mathrm{ecc}(c) = 11.5$ and c has three farthest leaves: $\bar{c}_1$ and $\bar{c}_2$ in T_1; and $\bar{c}_3$ in T_2. According to Lemma 1, $\bar{c}_3$ is farthest from every point on sub-tree T_1, whereas $\bar{c}_1$ and $\bar{c}_2$ are farthest from every point on sub-tree T_2.

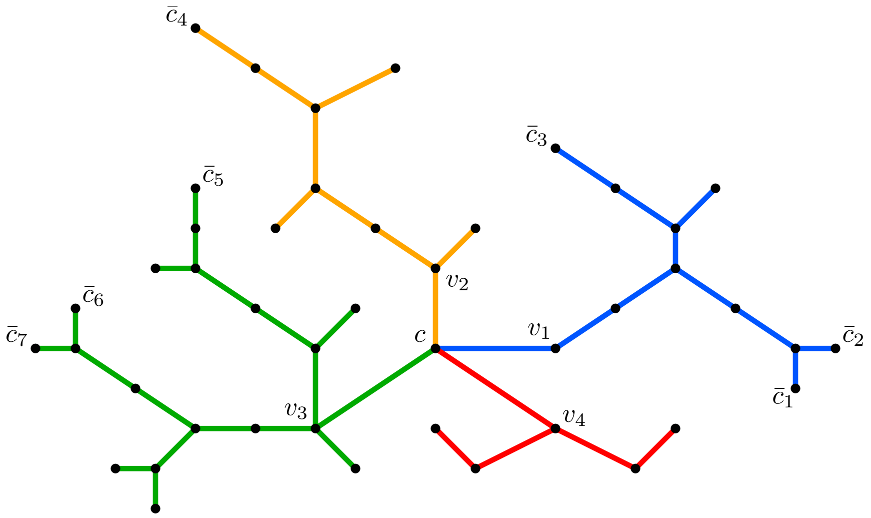


Figure 3: A tree T where all edges have unit weight. The absolute center c is located at a vertex and splits T into four sub-trees T_1 (blue), T_2 (orange), T_3 (green), and T_4 (red). The farthest distance from c is $\mathrm{ecc}(c) = 6$ and c has seven farthest leaves: $\bar{c}_1, \bar{c}_2$, and $\bar{c}_3$ in T_1; $\bar{c}_4$ in T_2; $\bar{c}_5$, $\bar{c}_6$, and $\bar{c}_7$ in T_3; and none in T_4. Let $L_1 = \{\bar{c}_1, \bar{c}_2, \bar{c}_3\}$, $L_2 = \{\bar{c}_4\}$, $L_3 = \{\bar{c}_5, \bar{c}_6, \bar{c}_7\}$, and $L_4 = \emptyset$. According to Lemma 1, the leaves in $L_2 \cup L_3 \cup L_4$ are farthest from all points on T_1, the leaves in $L_1 \cup L_3 \cup L_4$ are farthest from all points on T_2, and the leaves in $L_1 \cup L_2 \cup L_4$ are farthest points from all points on T_3. All points on the red sub-tree T_4 share their farthest points with the absolute center c, since $L_4 = \emptyset$.

We split a tree T at its absolute center c into sub-trees as follows. When c lies on an edge uv with $u \neq c \neq v$, we split T into two sub-trees: the sub-tree T_1 containing the sub-edge uc, and the sub-tree T_2 containing the sub-edge cv. When c lies on a vertex with neighbors $v_1, v_2, \ldots, v_r$, we split T into r sub-trees T_1, T_2, $\ldots$, T_r, where sub-tree T_i contains the sub-edge cv_i. Figures 2 and 3 demonstrate splitting trees at their absolute center and illustrate the following lemma that relates absolute centers with farthest points.

Lemma 1 *Let T be a tree with absolute center c, let T_i be one of the sub-trees obtained by splitting T at c, and let p be a point on T_i with $p \neq c$. The farthest distance from p in T is $\operatorname{ecc}(p) = d(p,c) + \operatorname{ecc}(c)$ and the farthest points from p are precisely the farthest leaves from c outside of T_i.*

Proof: We show that every path from p to a farthest leaf $\bar{p}$ passes through c.

Let l and $\bar{l}$ be two most distant leaves of T. According to Theorem 1, c is located midway along the path from l to $\bar{l}$ with $\operatorname{ecc}(c) = d(c,l) = d(c,\bar{l})$. When splitting T at c, the leaves l and $\bar{l}$ end up in different sub-trees. Assume, without loss of generality, that l and p lie in different sub-trees, which implies that c lies on a path from p to l, i.e., $d(p,l) = d(p,c) + d(c,l)$.

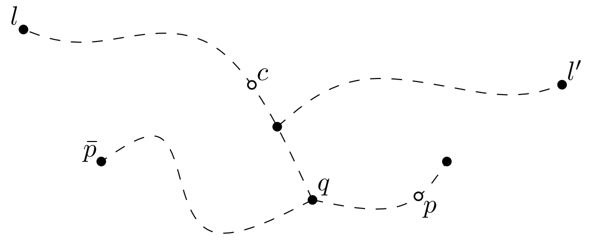


Figure 4: The impossible layout of the paths in a tree T where a point p on T has a farthest point $\bar{p}$ such that the shortest path from p to $\bar{p}$ avoids the absolute center c of T. According to Theorem 1, the absolute center c is located midway on a path between two most distant leaves l and $\bar{l}$. The path from p to one of these leaves—l in this case—passes through c.

Assume, for the sake of a contradiction, that p has a farthest leaf $\bar{p}$ and the path in T from p to $\bar{p}$ avoids c, i.e., $\bar{p}$ is in the same sub-tree as p. Let q be the meeting point of the path from c to $\bar{p}$ with the path from p to $\bar{p}$. Figure 4 shows this (impossible) constellation in which we have

$$d(p,q) + d(q,\bar{p}) = d(p,\bar{p}) = \operatorname{ecc}(p) \geq d(p,l) = d(p,q) + d(q,l) \ ,$$

which implies $d(q,\bar{p}) \geq d(q,l)$, i.e., l is no farther away from q than $\bar{q}$.

Since q lies on the path from p to $\bar{p}$ and this path avoids c, we have $q \neq c$ and, thus, $d(c,q) > 0$. We arrive at a contradiction via

$$d(c,\bar{p}) = d(c,q) + d(q,\bar{p}) > d(q,\bar{p}) \geq d(q,l) = d(q,c) + d(c,l) > d(c,l) \ ,$$

which implies that $\bar{p}$ is farther away from c than the farthest leaf l.

Therefore, every path from p to any farthest leaf must contain the absolute center c of T. The farthest distance we can travel along a path from p through c is $d(p,c)+d(c,l) = d(p,c)+\mathrm{ecc}(c)$. Conversely, every farthest leaf from c outside of T_i achieves this distance and, thus, is farthest from p. $\square$

We perform eccentricity queries on tree networks as follows. Let T be a tree with absolute center c. Consider an eccentricity query from a point p on edge uv where u is closer to c than v, i.e., $d(c,u) < d(c,v)$. The shortest path from p to c leads through u, i.e., $d(p,c) = w_{pu} + d(u,c)$, and we have $\mathrm{ecc}(p) = d(p,c) + \mathrm{ecc}(c) = w_{pu} + d(u,c) + \mathrm{ecc}(c)$. Thus, we can determine the eccentricity of p in constant time, provided that we know the eccentricity of c and the network distance from c to every vertex of T.

We perform farthest-point queries on T as follows. Let T_1, T_2, $\ldots$, T_r be the sub-trees obtained by splitting T at c. For $i = 1, 2, \ldots, r$, let L_i denote the set of farthest leaves from c in sub-tree T_i, i.e., $L_i \coloneqq \{l \in T_i \mid \mathrm{ecc}(c) = d(c,l)\}$. For a farthest-point query from the absolute center c, we report all leaves in L_i for all $i = 1, 2, \ldots, r$. For a farthest-point query from a point p on sub-tree T_i with $p \neq c$, we report all leaves in each L_j with $j \neq i$ as farthest points of p. For a query point with k farthest points, this takes $O(k)$ time, provided that we know the sets of farthest leaves $L_1, L_2, \ldots, L_r$ and provided that we can identify the sub-tree among $T_1, T_2, \ldots, T_r$ containing the query point in constant time.

The description of eccentricity queries and farthest-point queries on trees suggests which auxiliary data should be pre-computed. For a given tree T with n vertices, we first locate the absolute center c of T in $O(n)$ time using Handler's Algorithm [14]. Using a breadth-first-search from c, we perform three tasks: we compute the distances from c to every vertex of T, we label each edge with an index indicating its sub-tree, and we determine the farthest leaves in each sub-tree, which yields the sets $L_1, L_2, \ldots, L_r$. Altogether, we spend $O(n)$ time to obtain our data structure, which is summarized in the following theorem.

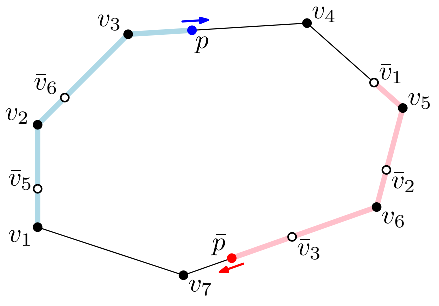
Theorem 2 (Tree Data Structure) *Let T be a tree network with n vertices. There is a data structure with $O(n)$ construction time supporting eccentricity queries on T in constant time and farthest-point queries on T in $O(k)$ time, where k is the number of farthest-points.*

By storing the lengths of the lists $L_1, L_2, \ldots, L_r$ alongside with their sum $\sigma = \Sigma_{i=1}^{r} |L_i|$, the data structure from Theorem 2 also allows us to count the number of farthest points from any query point in a tree network in constant time: the number of farthest points from the absolute center c is σ and the number of farthest points from a query point $p \neq c$ in sub-tree T_i is $\sigma - |L_i|$.

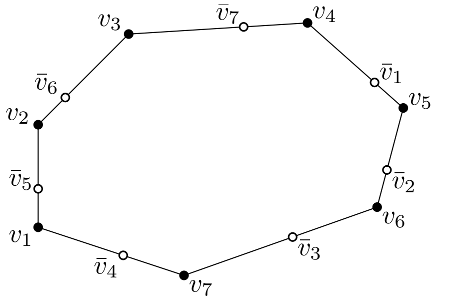
2.2 Cycles

Let C be a cycle network and let w_C be the sum of all edge weights of C. Each point p on C has exactly one farthest point $\bar{p}$ located on the opposite side of C with $\mathrm{ecc}(p) = d(p,\bar{p}) = w_C/2$. We call $\bar{p}$ the *antipodal* of p in C. Supporting eccentricity queries on C amounts to calculating and storing the value $w_C/2$.

To support farthest-point queries, we subdivide C at the antipodal of each vertex and store a pointer from each vertex to its antipodal and vice versa. To compute this subdivision, we first locate the antipodal $\bar{v}$ for some initial vertex v by walking a distance of $w_C/2$ from v along C. As illustrated in Figure 5, we then sweep a point p from the position of v to the position of $\bar{v}$ along C while maintaining the antipodal $\bar{p}$. During this sweep we subdivide C at p whenever $\bar{p}$ hits a vertex and at $\bar{p}$ whenever p hits a vertex. The entire sweep takes linear time, thus, the resulting data structure occupies linear space.



(a) Sweeping along the cycle.



(b) The subdivision.

Figure 5: A sweep along cycle C starting from v_1 and the resulting subdivision. For instance, any point on sub-edge $v_5\bar{v}_2$ has its antipodal on sub-edge $\bar{v}_5v_2$.

Using the subdivided network, we answer farthest-point queries as follows. For a query point p on edge uv of C, we first locate the sub-edge ab containing p using binary search. This takes $O(\log n)$ time for a cycle with n vertices, since we subdivide uv at most n times. Let $\bar{a}$ and $\bar{b}$ be farthest from a and b, respectively. The farthest point $\bar{p}$ from p is located on sub-edge $\bar{a}\bar{b}$ at distance w_{ap} from $\bar{a}$.

Lemma 2 (Cycle Data Structure) *Let C be a cycle network with n vertices. There is a data structure with construction time $O(n)$ supporting eccentricity queries on C in constant time and farthest-point queries on C in $O(\log n)$ time.*

One might wonder if there is a way to improve the time for farthest-point queries in Lemma 2. The following lemma shows that it is impossible, unless we make additional assumptions about the edge weights or our model of computation.

Lemma 3 *Answering farthest-point queries on a cycle with n vertices requires $\Omega(\log n)$ comparisons for any data structure supporting such queries.*

Proof: We show the claim via a reduction from the static predecessor search problem. Let $x_1, x_2, \ldots, x_n$ be real values with $0 < x_1 < x_2 < \cdots < x_n < 1$. Determining the predecessor of any query value $\xi \in (0, 1)$ among $x_1, x_2, \ldots, x_n$ requires $\Omega(\log n)$ comparisons for any data structure supporting such queries.

We construct a cycle from our instance of predecessor search, as illustrated in Figure 6. We begin with two vertices u and v connected by an edge of weight one. We subdivide this edge at the points p_i with distance x_i from u for $i = 1, 2, \ldots, n$. Finally, we add another edge e from u to v of weight one to complete the cycle.

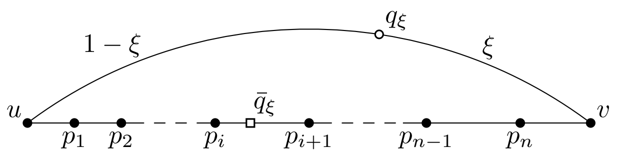


Figure 6: Reducing a predecessor search to a farthest-point query on a cycle.

To answer a predecessor query for ξ, we perform a farthest-point query from the point q_ξ with distance ξ from v on the edge e. The antipodal $\bar{q}_\xi$ of q_ξ has distance ξ from u. Thus, the endpoint of the edge $p_i p_{i+1}$ containing $\bar{q}_\xi$ that is closer to u corresponds to the predecessor of ξ among x_1, x_2, $\ldots$, x_n.

There can be no data structure supporting farthest-point queries on C in $o(\log n)$ time, since it would support predecessor queries in $o(\log n)$ time. □

The bound from Lemma 3 extends to uni-cyclic and cactus networks and, thereby, justifies the claim that our data structures achieve optimal query times.

2.3 Uni-Cyclic Networks

A network with exactly one simple cycle is called *uni-cyclic* [13]. As illustrated in Figure 7, every uni-cyclic network U consists of a cycle C with trees $T_1, T_2, \ldots, T_l$ attached to C at vertices $v_1, v_2, \ldots, v_l$, respectively. We refer to each tree T_i as a *branch* of U and to the vertex v_i as the *hinge* of branch T_i.

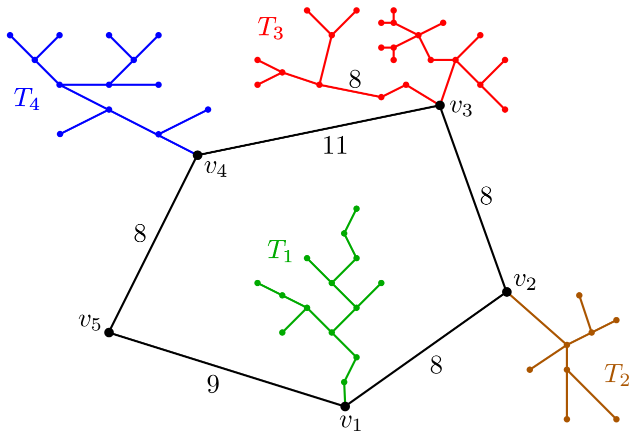
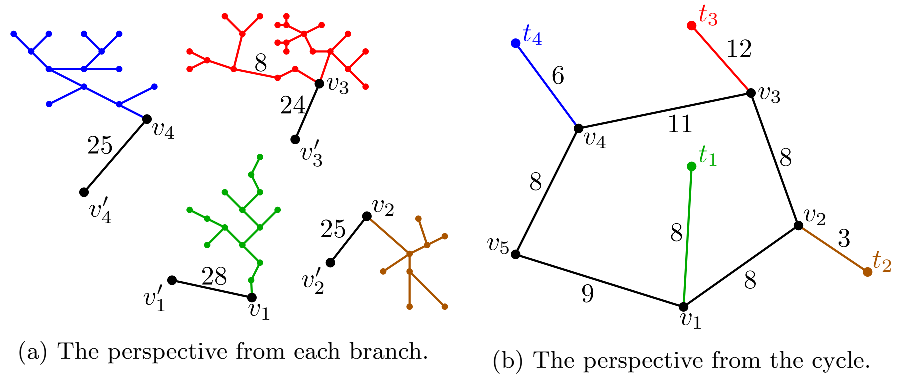


Figure 7: A uni-cyclic network with four branches. Edges have weight one whenever no weight is indicated.

Our data structure for uni-cyclic networks consists of two adjoint data structures: one for queries from the branches and one for queries from the cycle. These data structures are based on the following contractions. For each branch T with hinge v in a uni-cyclic network U, we represent the longest paths in U starting at hinge v and leading out of T by an edge vv' (to a dummy vertex v') of weight $w_{vv'} = \mathrm{ecc}_{U \setminus T}(v)$. We call the tree consisting of T and the edge vv' the *perspective* of T from U denoted by $\mathrm{per}_U(T)$. Similarly, we define the *perspective* of U from the cycle C denoted by $\mathrm{per}_U(C)$ as the network consisting of C with an additional edge vt to a dummy vertex t for each branch T with hinge v. The edge vt represents the paths starting from hinge v leading into a branch T and, thus, has weight $w_{vt} = \mathrm{ecc}_T(v)$. Figure 8 summarizes the different perspectives for the network from Figure 7.



(a) The perspective from each branch. (b) The perspective from the cycle.

Figure 8: Viewing the network in Figure 7 from the branches (a) and the cycle (b).

The data structure for queries from the branches $T_1, T_2, \ldots, T_l$ consists of the tree data structures for $\mathrm{per}(T_1)$, $\mathrm{per}(T_2)$, $\ldots$, $\mathrm{per}(T_l)$ from Section 2.1. Consider a point p on branch T_i. An eccentricity query from p in $\mathrm{per}(T_i)$ yields the eccentricity of p in U. A farthest-point query from p in $\mathrm{per}(T_i)$ reports the farthest points from p that are located in T_i and this query reports the dummy vertex v_i' whenever p has farthest points outside of T_i. As we shall explain later, we obtain the farthest points from p outside of T_i using a query from the hinge of T_i in our data structure for the cycle perspective $\mathrm{per}(C)$.

The data structure for queries from the cycle C consists of two components: the first component reports the farthest points from C on C itself, using the data structure for cycles from Section 2.2 on C ignoring the remainder of U. The second component reports which branches among $T_1, T_2, \ldots, T_l$, if any, contain farthest points by supporting queries for the farthest among the branch-representing dummy vertices $t_1, t_2, \ldots, t_l$ from any query point on C in the cycle perspective $\mathrm{per}(C)$. We call this type of query a *farthest-branch query.*

2.3.1 Farthest-Branch Queries

We consider the perspective $\mathrm{per}_U(C)$ for the cycle C of a uni-cyclic network. The vertices $t_1, t_2, \ldots, t_l$ represent the compressed branches of U in $\mathrm{per}_U(C)$ and are connected to the hinges v_1, v_2, …, v_l, respectively. Furthermore, let $\bar{v}_i$ denote the farthest point on C from hinge v_i, i.e., $d_C(v_i, \bar{v}_i) = \mathrm{ecc}_C(v_i)$.

We call a vertex t_i *relevant* if there exists a point p on C that has t_i as a farthest vertex among $t_1, t_2, \ldots, t_l$, and we call t_i *irrelevant* otherwise. Knowing which of $t_1, t_2, \ldots, t_l$ are relevant allows us to perform farthest-branch queries, i.e., report all branches containing farthest points from a query point on C.

Lemma 4 *The vertex t_i is relevant if and only if t_i is a farthest vertex from $\bar{v}_i$ among $t_1, t_2, \ldots, t_l$, i.e., $d(t_i, \bar{v}_i) = \max_{j=1}^{l} d(t_j, \bar{v}_i)$.*

Proof: We show both directions via indirection. When t_i is irrelevant no point on C—including $\bar{v}_i$—has t_i as a farthest vertex among $t_1, t_2, \ldots, t_l$.

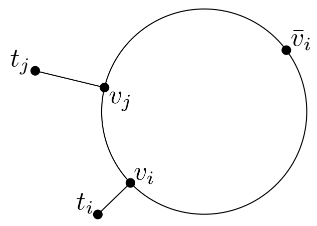


Figure 9: The constellation where v_j lies on the clockwise path from v_i to $\bar{v}_i$.

Conversely, let there be some vertex t_j that is farther away from $\bar{v}_i$ than t_i, i.e., $d(\bar{v}_i, t_i) < d(\bar{v}_i, t_j)$. As illustrated in Figure 9, v_j lies either on the clockwise or counter-clockwise path from t_i to $\bar{v}_i$. Since either path is a shortest path, we have $d(t_i, \bar{v}_i) = d(t_i, v_j) + d(v_j, \bar{v}_i)$. Thus, t_j is farther from v_j than t_i, as

$$d(t_i, v_j) = d(t_i, \bar{v}_i) - d(\bar{v}_i, v_j) < d(t_j, \bar{v}_i) - d(\bar{v}_i, v_j) = d(t_j, v_j) \ .$$

Now t_i is irrelevant, as t_j is farther from any point p on C than t_i, since

$$d(p, t_i) \leq d(p, v_j) + d(v_j, t_i) < d(p, v_j) + d(v_j, t_j) = d(p, t_j) \ .$$

□

According to Lemma 4, a vertex t_i is irrelevant when there is some other vertex t_j with $d(t_i, \bar{v}_i) < d(t_j, \bar{v}_i)$. In this case, we say that t_i *is dominated by* t_j and write $t_i \prec t_j$. Algorithm 1 below computes the relevant vertices using dominance. We begin with a circular list containing the vertices $t_1, t_2, \ldots, t_l$ in the order as the hinges $v_1, v_2, \ldots, v_l$ appear along the cycle C. We traverse the list in counterclockwise order and delete vertices whenever they are dominated by their successor (SUCC) or their predecessor (PRED). We mark a vertex t as processed if we can neither remove t nor any of its neighbors based on this criteria. As we shall see in the proof of Theorem 3, each irrelevant vertex t becomes the neighbor of some vertex dominating t throughout this process. Therefore, considering only immediate neighbors is sufficient.

Algorithm 1: Determining the relevant vertices

```
input  : The vertices $t_1, t_2, \ldots, t_l$ stored in a circular list.
output : The relevant vertices among $t_1, t_2, \ldots, t_l$.
1  Mark each $t_1, t_2, \ldots, t_l$ as unprocessed;
2  $t \leftarrow t_1$;
3  while $t$ is unprocessed do
4      if $t \prec \textsc{pred}(t)$ or $t \prec \textsc{succ}(t)$ then
5          $t \leftarrow \textsc{succ}(t)$;
6          $\textsc{delete}(\textsc{pred}(t))$;
7      else if $\textsc{pred}(t) \prec t$ then $\textsc{delete}(\textsc{pred}(t))$;
8      else if $\textsc{succ}(t) \prec t$ then $\textsc{delete}(\textsc{succ}(t))$;
9      else /* $t \not\prec \textsc{pred}(t) \not\prec t \not\prec \textsc{succ}(t) \not\prec t$ */
10         Mark $t$ as processed;
11         $t \leftarrow \textsc{succ}(t)$;
12     end
13 end
```

Invariant 1 *The following invariants hold whenever Algorithm 1 marks a vertex as processed in Line 10.*

(i) Every marked vertex t dominates none of its current neighbors, i.e., we have $\textsc{pred}(t) \not\prec t$ and $\textsc{succ}(t) \not\prec t$.

(ii) Every marked vertex t is dominated by none of its current neighbors, i.e., we have $t \not\prec \textsc{pred}(t)$ and $t \not\prec \textsc{succ}(t)$.

Proof: Assume, for the sake of a contradiction, that there is a marked vertex t_i with a neighbor t_j such that $t_i \prec t_j$ or $t_j \prec t_i$ when Line 10 is executed. Assume, without loss of generality, that t_j has been marked after t_i or that t_j has never been marked so far. When we marked t_i as processed, t_i and t_j were not neighbors. When t_j became a neighbor of t_i, the previous neighbor t_k of t_i was either deleted in Line 6 with t_j assuming the role of variable t in Line 5 or t_k was deleted in Lines 8 with $t = t_j$. In both cases variable t would be at t_j with t_i as direct neighbor and we would have deleted t_i in Line 7 or in Line 8 before marking any other vertex as processed. Therefore, we have never marked t_i, which contradicts our assumption. □

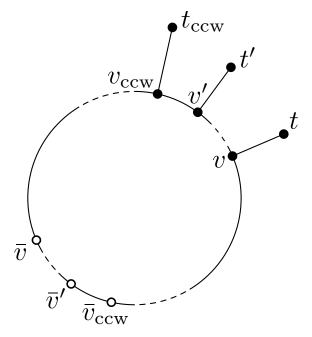
We pre-compute the distances from v_1 to all other vertices along C while constructing $\mathrm{per}(C)$. This allows us to compare distances along C in constant time and, thus, enables us to determine in constant time whether one vertex dominates another. We are prepared to proof the correctness of Algorithm 1.

Theorem 3 *Let S be the perspective of a uni-cyclic network U from its cycle C, and let $t_1, t_2, \ldots, t_l$ be the vertices of S representing the l branches of U given in clockwise order. Algorithm 1 computes all relevant vertices among $t_1, t_2, \ldots, t_l$ in $O(l)$ time, provided that checking for dominance takes constant time.*

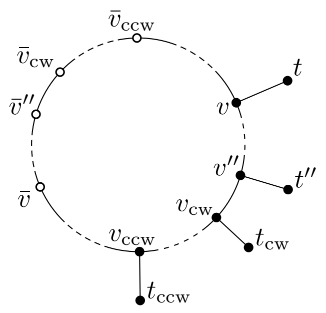
Proof: In each iteration of the while-loop of Algorithm 1, we either delete some vertex or we mark the vertex stored in t as processed ensuring that it will never assume the role of t again. Therefore, Algorithm 1 terminates in $O(l)$ steps, provided that checking for dominance is a constant time operation.

Algorithm 1 never deletes a relevant vertex, since we only delete vertices that are dominated and, thus, irrelevant. Consider the circular list of those vertices that remain after Algorithm 1 terminates. We assume, for the sake of a contradiction, that this list contains irrelevant vertices.

Let t be a relevant vertex and let t_{cw} be the first irrelevant vertex in clockwise direction from t, and let t_{ccw} be the first irrelevant vertex in counterclockwise direction from t. Since no vertex in the final list dominates any of its neighbors by Invariant 1, the final list contains at least four vertices with at least one vertex between t and t_{cw} and at least one vertex between t and t_{ccw}. The argument below remains valid when t_{cw} and t_{ccw} coincide. Let $\bar{v}$, $\bar{v}_{\mathrm{cw}}$ and $\bar{v}_{\mathrm{ccw}}$ be the farthest point on C from t, t_{cw}, and t_{ccw}, respectively. We distinguish the two cases illustrated in Figure 10, based on the relative positions of t, $\bar{v}$, and $\bar{v}_{\mathrm{ccw}}$.



(a) The point $\bar{v}_{\mathrm{ccw}}$ lies clockwise between t and $\bar{v}$.



(b) The point $\bar{v}_{\mathrm{ccw}}$ lies counterclockwise between t and $\bar{v}$.

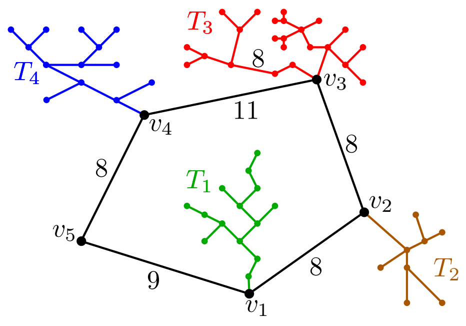
Figure 10: The cyclic order of the vertices t, t_{ccw}, t_{cw}, t', t'', and their corresponding antipodal points in the two cases from the proof of Theorem 3.

Consider the case when the point $\bar{v}_{\mathrm{ccw}}$ lies on the clockwise path from t to $\bar{v}$, as illustrated in Figure 10a. Let $t' := \textsc{pred}(t_{\mathrm{ccw}})$ be the clockwise neighbor of t_{ccw} and let $\bar{v}'$ be the farthest point from t' on C. The vertices $\bar{v}_{\mathrm{ccw}}$, $\bar{v}'$, and $\bar{v}$ appear clockwise in this order along C, which implies that $\bar{v}_{\mathrm{ccw}}$ lies on a shortest path from t to $\bar{v}'$, i.e., $d(t,\bar{v}') = d(t,\bar{v}_{\mathrm{ccw}}) + d(\bar{v}_{\mathrm{ccw}},\bar{v}')$. Furthermore, we have $t_{\mathrm{ccw}} \prec t$ and $t' \not\prec t$, since t_{ccw} was the first dominated vertex in clockwise direction from t. Together, this yields

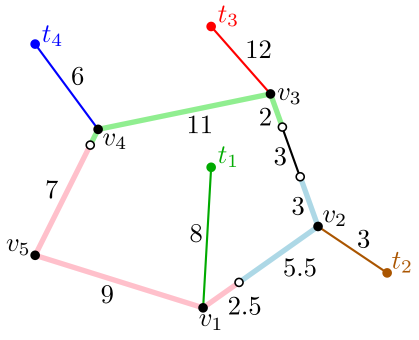
$$\begin{aligned} d(t_{\mathrm{ccw}},\bar{v}_{\mathrm{ccw}}) &\overset{t_{\mathrm{ccw}}\prec t}{<} d(t,\bar{v}_{\mathrm{ccw}}) = d(t,\bar{v}') - d(\bar{v}',\bar{v}_{\mathrm{ccw}}) \\ &\overset{t'\not\prec t}{\leq} d(t',\bar{v}') - d(\bar{v}',\bar{v}_{\mathrm{ccw}}) = d(t',\bar{v}_{\mathrm{ccw}}) \ , \end{aligned}$$

which implies $d(t_{\text{ccw}}, \bar{v}_{\text{ccw}}) < d(t', \bar{v}_{\text{ccw}})$, i.e., $t_{\text{ccw}} \prec t'$. This contradicts Invariant 1, because t_{ccw} remained in the circular list when Algorithm 1 terminated and, thus, t_{ccw} must be marked, yet t_{ccw} is dominated by its neigbhour t'.

Consider the case when $\bar{v}_{\text{ccw}}$ lies on the counterclockwise path from t to $\bar{v}$, as illustrated in Figure 10b. This implies that the vertices t, t_{cw}, t_{ccw}, and $\bar{v}$ appear clockwise in this order (with potentially $t_{\text{cw}} = t_{\text{ccw}}$). Therefore, $\bar{v}_{\text{cw}}$ also lies on the counterclockwise path from t to $\bar{v}$. Similarly, to the above, we derive the contradiction $t_{\text{cw}} \prec t''$ where $t'' := \textsc{succ}(t_{\text{cw}})$ is the counterclockwise neighbor of t_{cw}. Therefore, t_{cw} (and thus t_{ccw}) cannot exist and there are no irrelevant vertices in the circular list produced by Algorithm 1. □



(a) A uni-cyclic network.



(b) The farthest-branch subdivision.

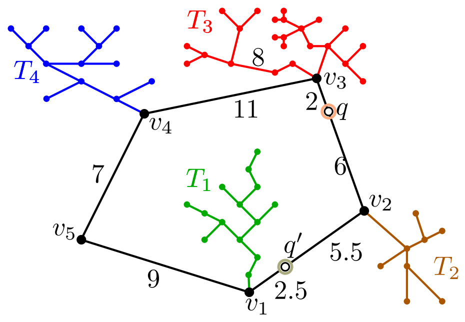
Figure 11: The farthest-branch subdivision (b) of the uni-cyclic network from (a). The sub-edges in (b) are shaded in a colour matching their farthest branch. On black sub-edges, no branch contains farhtest points—instead, the farthest point lies on the opposite side of the cycle. In this example, no points on the cycle have farthest points in branch T_2, i.e., t_2 is irrelevant.

We have a circular list of the relevant vertices among $t_1, t_2, \ldots, t_l$. Using this list, we pre-process S to support farthest-branch queries as follows. We pick any relevant vertex t_i and traverse C counterclockwise starting from $\bar{v}_i$ keeping track of the farthest branch. Since t_i and its counterclockwise successor $t_j = \textsc{succ}(t_i)$ are relevant, $\bar{v}_i$ has t_i as its farthest branch and $\bar{v}_j$ has t_j as its farthest branch. At some point p_{ij} between $\bar{v}_i$ and $\bar{v}_j$ the farthest branch changes from t_i to t_j; we subdivide C at p_{ij} and store t_i as farthest branch for the (sub)edges from $\bar{v}_i$ to p_{ij}. We continue subdividing C in this fashion into at most l chains, one for each relevant vertex. Using a binary search on these chains, we answer farthest-branch queries in $O(\log l)$ time. Figure 11 illustrates an example of this subdivision for the network from Figure 7. As in all figures in this section, edges have unit weight unless indicated otherwise.

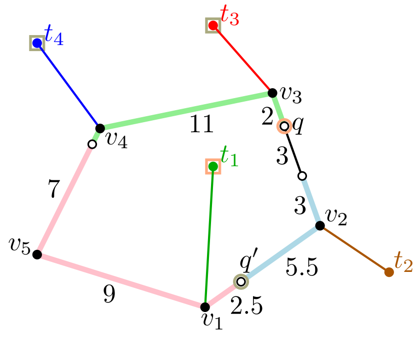
Theorem 4 *Let U be a uni-cyclic network with n vertices and cycle C of length l. There is a data structure with construction time $O(n)$ supporting farthest-branch queries from any query point on C in $O(\log l)$ time.*

2.3.2 Queries in Uni-Cyclic Networks

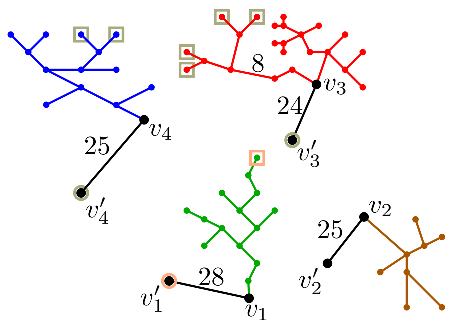
We perform an eccentricity query from a point q on U as follows. When q lies on some branch T, we use the tree data structures for the perspective $T' = \text{per}(T)$ from T, as $\text{ecc}_{T'}(q) = \text{ecc}_U(q)$ by construction of T'. When q lies on the cycle C, we first compute the distance $d_B := \max_{i=1}^{l} d(q, t_i)$ from q to the farthest vertex among $t_1, t_2, \ldots, t_l$ in the perspective $S = \text{per}(C)$ from C, using our data structure for farthest-branch queries in S. Then, we compute the farthest distance $\text{ecc}_C(q)$ from q on C, using the cycle data structure for C. The greater distance of the two is the eccentricity of q in S and, thus, the eccentricity of q in U, i.e., $\text{ecc}_U(q) = \text{ecc}_S(q) = \max\{d_B, \text{ecc}_C(q)\}$. This way, we answer eccentricity queries from branches in constant time and from the cycle in $O(\log l)$ time.



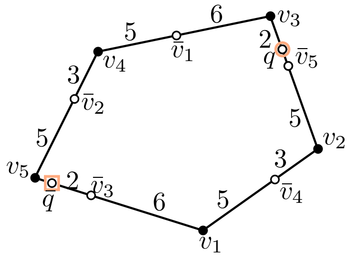
(a) Two queries form the cycle.



(b) Farthest-branch queries for q and q'.



(c) Cascading the queries into the branches.

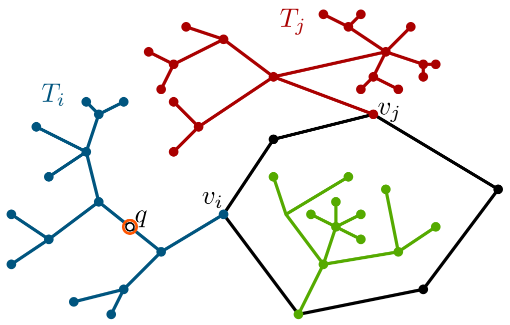


(d) Querying for the antipodal of q.

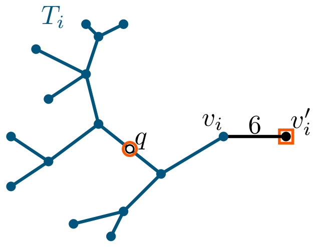
Figure 12: The farthest-point queries from the points q (salmon) and q' (ocker) on the cycle of a uni-cyclic network (a). We perform a farthest-branch query for both query points (b) and then cascade the queries into the perspectives from the branches (c) and into the cycle (d) as needed.

We perform a farthest-point query from a point q on the cycle C of U as illustrated in Figure 12. We first determine the farthest distance from q to any point in a branch of U with a farthest-branch query from q in S and we determine the farthest distance from q to any point on C with a corresponding eccentricity query. Comparing these distances reveals where we find the farthest points from q in U. We then report the antipodal of q on C if it is farthest and then cascade the query in each branch containing farthest points. More precisely, if T was a branch containing farthest points from q then perform a farthest-point query from v' in the perspective $T' = \mathrm{per}_U(T)$ of U from T, where v' is the vertex representing the exterior of T. The farthest leaves from v' in T' are also farthest from any point outside of T that has farthest points in T.

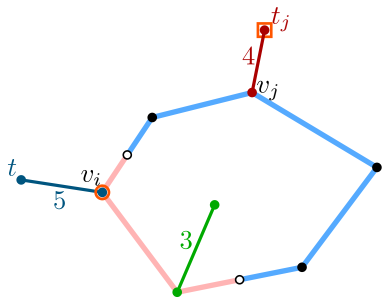
We perform a farthest-point query from a point q on a branch T of U with hinge v as follows. We begin with a farthest-point query in the perspective $T' = \mathrm{per}(T)$ of U from T. This reports all farthest points from q in T and, potentially, the point v' representing the exterior of T. When cascading the query from T' into the perspective $S = \mathrm{per}(C)$ of U from the cycle C, we need to perform a farthest-point query in S from the vertex t representing T. However, we have no structure to support this query directly. Instead, we query from v in S, which leads to a *good case* and a *bad case*.



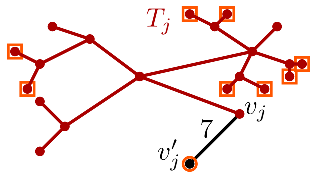
(a) A query from a branch T_i.



(b) Query in the perspective from T_i.

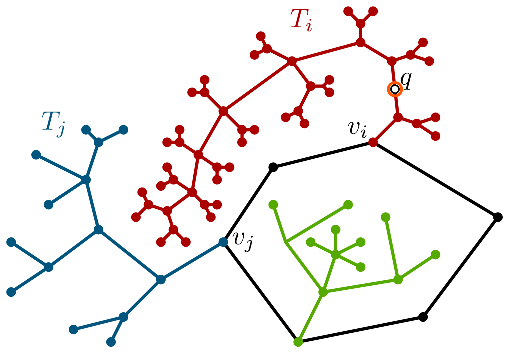


(c) Cascading the query into per(C).

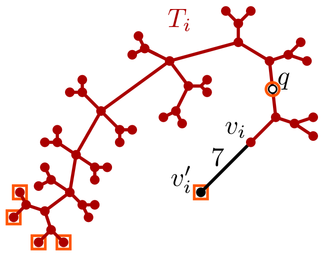


(d) Cascading the query into branch T_j.

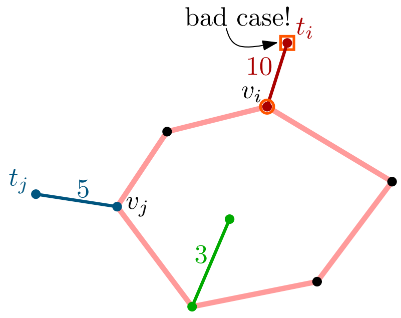
Figure 13: A farthest-point query from a branch T_j of a unicyclic network (a). We perform a farthest-point query in the perspective from the branch containing the query point (b). We cascade the query into the perspective from the cycle (c) and then into another branch T_j as needed (d).



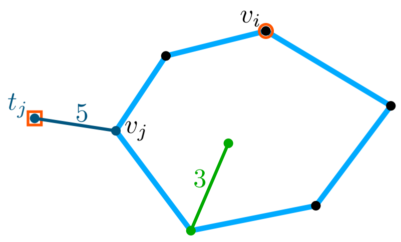
(a) A query from a branch T_i.



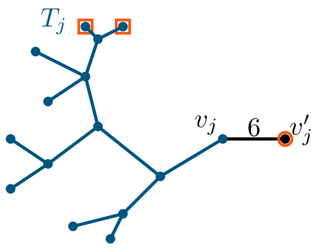
(b) Query in the perspective from T_i.



(c) Farthest-branch query leads back into T_i.



(d) Fixed farthest-branch query.



(e) Query continues in correct branch T_j.

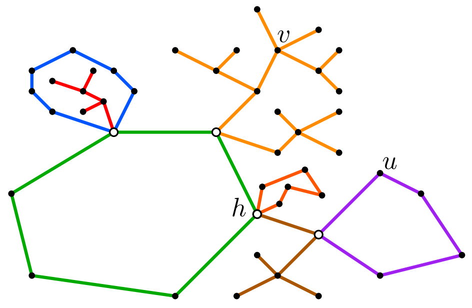
Figure 14: The bad case for a farthest-point query from a branch T_i (red) of a unicyclic network (a). We perform a farthest-point query in the perspective from the branch containing the query point (b). We cascade the query into the perspective from the cycle (c), however, this query sends us back into the red branch, since the red branch is farthest on the entire cycle. To fix this, we remove the edge $t_i v_i$ representing the red branch T_i (d) and repeat the query in the resulting network. This time, we obtain the correct branch T_j where we cascade the query (e) to determine the farthest points in this (blue) branch. Observe that the fixed network (d) is irrelevant for queries from other branches which will use the normal perspective from the cycle (c).

In the good case, shown in Figure 13, v has some farthest point in S other than t, which means that the farthest points from t are the farthest points from v (potentially excluding t itself). In the bad case, shown in Figure 14, t is the only farthest point from v in S. Fortunately, the bad case can only appear for exactly one branch of U, as t being the only farthest vertex from v implies that t is farthest from all other vertices of S as well. Therefore, we can deal with the bad case by computing the farthest points from v in the network $S - vt$ (during the preprocessing phase) and storing the result with v. We use this only when cascading a query from T into S, i.e., when we know that there are farthest points from the query point q outside of T.

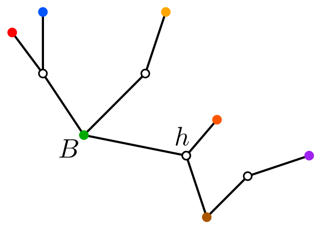
Theorem 5 (Uni-Cyclic Data Structure) *Let U be a uni-cyclic network with n vertices and a cycle of length l. There is a data structure with size and construction time $O(n)$ supporting eccentricity queries from the branches of U in $O(1)$ time, eccentricity queries from the cycle in $O(\log l)$ time, farthest-point queries from branches in $O(k)$ time, and farthest-point queries from the cycle in $O(k + \log l)$ time, where k is the number of farthest-points.*

3 Cactus Networks

In this section, we construct a data structure supporting eccentricity queries and farthest-point queries on cactus networks. Recall that a cactus network is a network in which no two simple cycles share an edge.



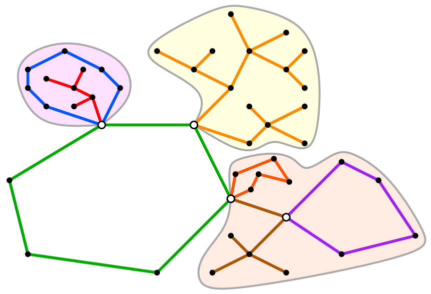
(a) A cactus network G with 7 bags.

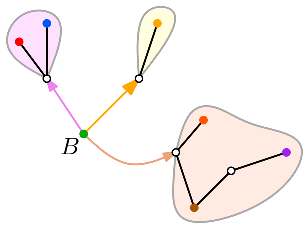


(b) The tree structure $T(G)$ of G.

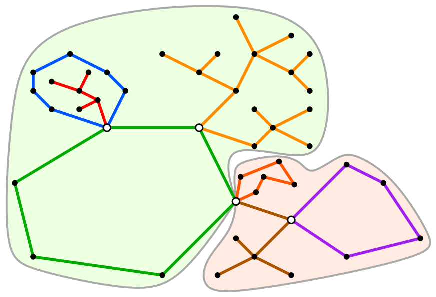
Figure 15: A cactus network (a) with its tree structure (b). The blocks and branches are indicated in colours and the hinges connecting them are marked as empty discs. For example, the green edges form a block and the red edges form a branch. Vertex h is a hinge, because h is a cut-vertex contained in more than one bag; vertex v is a cut-vertex but not a hinge, since v is only contained in the yellow branch; and vertex u is not a cut-vertex and, thus, also not a hinge.

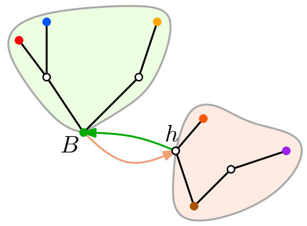
The following notions prove useful when describing cactus networks; examples are shown in Figure 15. A *cut-vertex* is a vertex whose removal increases the number of connected components, a *block* is a maximal connected sub-graph without cut-vertices and with at least three vertices, and a *branch* is a (maximal) tree that remains when removing the edges of all blocks and all resulting isolated vertices. We treat blocks and branches in a similar fashion, so we refer to a sub-network B as a *bag* when B is either a block or a branch. Decomposing a network with n vertices into its blocks and branches takes $O(n)$ time [17]. A *hinge* is a vertex contained in more than one bag. The *tree structure* of a cactus network G, denoted by $T(G)$, is the tree whose vertices are the hinges and bags of G where a hinge h is connected to a bag B when h lies in B [16].


(a) The bag-cuts of the green block.



(b) Arcs in $T(G)$ of the bag-cuts in (a).


(c) The bag-cut and the co-bag-cut of B.



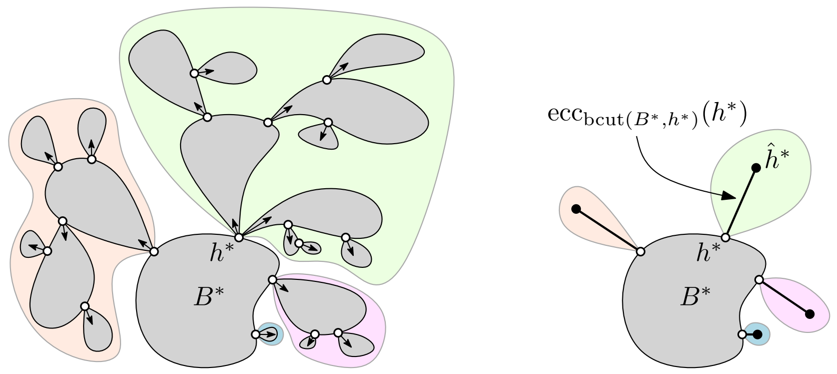
(d) Arcs in $T(G)$ of the cuts in (c).

Figure 16: The bag-cuts of the green block (a) together with their corresponding arcs (of matching colour) in the tree structure (b). The bag-cut (salmon) and co-bag-cut (light green) corresponding to the two arcs of an edge from B to h in the tree structure (d) and in the network itself (c).

Let B be a bag containing hinge h. The component containing h after removing all edges of B is called the *bag-cut* of B at h, denoted by $\text{bcut}(B, h)$. The component containing h after removing all edges of $\text{bcut}(B, h)$ is called the *co-bag-cut* of B at h, denoted by $\text{co-bcut}(B, h)$. Figure 16 gives examples of bag-cuts and co-bag-cuts for the cactus network from Figure 15. We view an (undirected) edge from B to h in the tree structure as two (directed) arcs $B \to h$ and $h \to B$. As illustrated in Figures 16c and 16d, we associate $\text{bcut}(B, h)$ with $B \to h$ and we associate $\text{co-bcut}(B, h)$ with $h \to B$. We use this correspondence as orientation. For example, we store the eccentricity of h in $\text{bcut}(B, h)$ with the arc $B \to h$ and the eccentricity of h in $\text{co-bcut}(B, h)$ with the arc $h \to B$.

3.1 Eccentricity Queries

To support eccentricity queries on a bag B of a network G, we compress the bag-cuts of B like we compress the branches of uni-cyclic networks: for any hinge $h \in B$ we replace $\mathrm{bcut}(B, h)$ with a vertex $\hat{h}$ and an edge $h\hat{h}$ whose weight is the largest distance from h to any point in $\mathrm{bcut}(B, h)$, i.e., $w_{h\hat{h}} = \mathrm{ecc}_{\mathrm{bcut}(B,h)}(h)$. We refer to the resulting network as the *perspective* of G from B, denoted by $\mathrm{per}_G(B)$. The perspective of G from bag B preserves farthest distances of G, i.e., we have $\mathrm{ecc}_{\mathrm{per}_G(B)}(p) = \mathrm{ecc}_G(p)$ for all p on B.

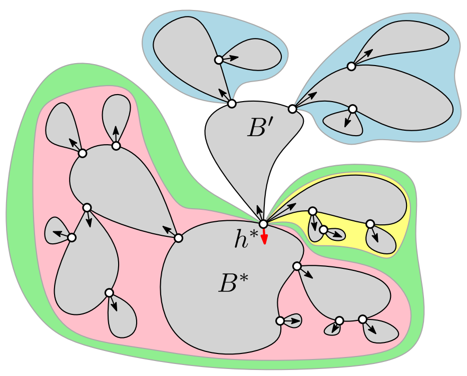


(a) The bag-cuts of the starting bag B^*. (b) The perspective from B^*.

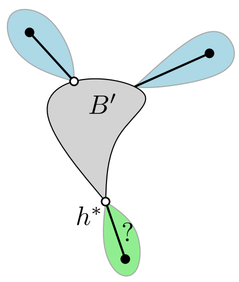
Figure 17: A schematic view of the construction of the perspective $\mathrm{per}(B^*)$ from the initial bag B^*. For every hinge h^* of B^*, the bag-cut $\mathrm{bcut}(B^*, h^*)$ is replaced with an edge $h^*\hat{h}^*$ weighted with the largest distance from h^* into $\mathrm{bcut}(B^*, h^*)$. The extended shortest path trees from the hinges of B^* are outlined as arrows from the hinges of the network. An arrow pointing into the co-bag-cut of bag B at hinge h means that we obtain the extended shortest path tree from h into $\text{co-bcut}(B, h)$ as a by-product of the construction of the perspective from B^*.

The perspective from a branch T is a tree, since $\mathrm{per}(T)$ results from attaching pendant edges to the tree T. The perspective from a block C of a cactus network is a uni-cyclic network, since $\mathrm{per}(C)$ results from attaching pendant edges to the cycle C. Since we already have efficient data structures for trees and uni-cyclic networks, the challenge lies in constructing these perspectives in linear time.

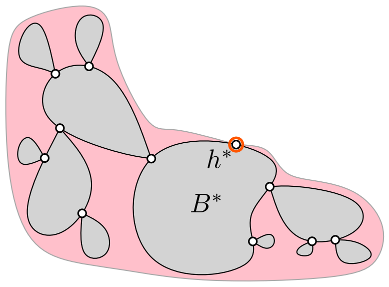
As illustrated in Figure 17, we first construct the perspective $\mathrm{per}(B^*)$ of an arbitrary bag B^* by computing the extended shortest path tree of each hinge h^* of B^* in $\mathrm{bcut}(B^*, h^*)$. This takes linear time using a modified breadth-first search from each hinge h^* where we cut each simple cycle C at the antipodal $\bar{v}$ of the vertex v where we first enter C. Using the data structure for queries in $\mathrm{per}(B^*)$ and the extended shortest path trees from the hinges of B^*, we then construct the perspectives from the bags adjacent to B^* and continue in a breadth-first search fashion. Next, we describe the propagation step.



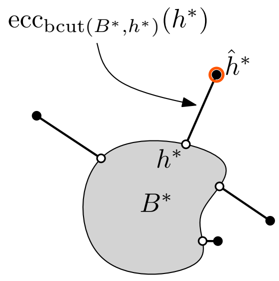
(a) The bag-cuts of neighboring bag B'.



(b) The missing weight in $\mathrm{per}(B')$.



(c) The desired query in $\text{co-bcut}(B^*, h^*)$.



(d) The actual query in $\mathrm{per}(B^*)$.

Figure 18: A schematic view of the construction of the perspective of a bag B' neighboring the initial bag B^*. From the construction of $\mathrm{per}(B^*)$, we already have the distance information for the bag-cuts (blue) of B' at all hinges except for the one at h^* (green) and we have the distances for all co-bag-cuts of neighbors of B' (yellow) other than $\text{co-bcut}(B^*, h^*)$ (red). To obtain the missing value $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ we would like to query from h^* in $\text{co-bcut}(B^*, h^*)$, but instead we perform a query from $\hat{h}^*$ in $\mathrm{per}(B^*)$ to avoid constructing a data structure for $\text{co-bcut}(B^*, h^*)$.

Let B' be a bag neighboring B^* at hinge h^*, as shown in Figure 18. For all hinges h' of B' with $h' \neq h^*$, the extended shortest path tree of h' in $\mathrm{bcut}(B', h')$ is a sub-tree of the extended shortest path tree of h^* in $\mathrm{bcut}(B^*, h^*)$. So we know the weight of the edge $\hat{h}'h'$ in $\mathrm{per}(B')$, where $\hat{h}'$ represents $\mathrm{bcut}(B', h')$. At hinge h^*, we have the extended shortest path trees of h^* in every co-bag-cut at h^* except for $\text{co-bcut}(B^*, h^*)$, the one leading back into B^*. The eccentricity of h^* in $\mathrm{bcut}(B', h^*)$ is the largest distance from h^* into any co-bag-cut $\text{co-bcut}(B, h^*)$ for all bags B neighboring B' at h^*, i.e.,

$$\mathrm{ecc}_{\mathrm{bcut}(B',h^*)}(h^*) = \max\left\{ \mathrm{ecc}_{\text{co-bcut}(B,h^*)}(h^*) \;\middle|\; \begin{array}{l} B \text{ is a bag with} \\ h^* \in B \neq B' \end{array} \right\} . \tag{1}$$

We need the value $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ to compute $\mathrm{per}(B')$. The difference between the perspective $\mathrm{per}_G(B^*)$ of G from B^* and the perspective $\mathrm{per}_{\text{co-bcut}(B^*,h^*)}(B^*)$ of co-bcut(B^*, h^*) from B^* is that the latter lacks the edge $\hat{h}^*h^*$ where $\hat{h}^*$ represents bcut(B^*, h^*) in $\mathrm{per}_G(B^*)$. We avoid constructing $\mathrm{per}_{\text{co-bcut}(B^*,h^*)}(B^*)$ using the following observation. The farthest points from $\hat{h}^*$ in $\mathrm{per}_G(B^*, h^*)$ are also farthest from h^* in $\mathrm{per}_{\text{co-bcut}(B^*,h^*)}(B^*) = \mathrm{per}_G(B^*, h^*) - \hat{h}^*h^*$, but they are closer by $w_{h^*\hat{h}^*}$, i.e.,

$$\begin{aligned}\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*) &= \mathrm{ecc}_{\mathrm{per}(B^*)-\hat{h}^*h^*}(h^*)\\ &= \mathrm{ecc}_{\mathrm{per}(B^*)}(\hat{h}^*) - w_{h^*\hat{h}^*}\\ &= \mathrm{ecc}_{\mathrm{per}(B^*)}(\hat{h}^*) - \mathrm{ecc}_{\mathrm{bcut}(B^*,h^*)}(h^*)\ .\end{aligned}$$

Consider the extended shortest path tree from $\hat{h}^*$ in $\mathrm{per}_G(B^*)$. Removing the edge $\hat{h}^*h^*$ from this tree yields the extended shortest path tree from h^* in $\mathrm{per}_G(B^*) - \hat{h}^*h^* = \text{co-bcut}(B^*, h^*)$. We obtain $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ with a constant time query from $\hat{h}^*$ in $\mathrm{per}_G(B^*)$, as illustrated in Figure 18d.

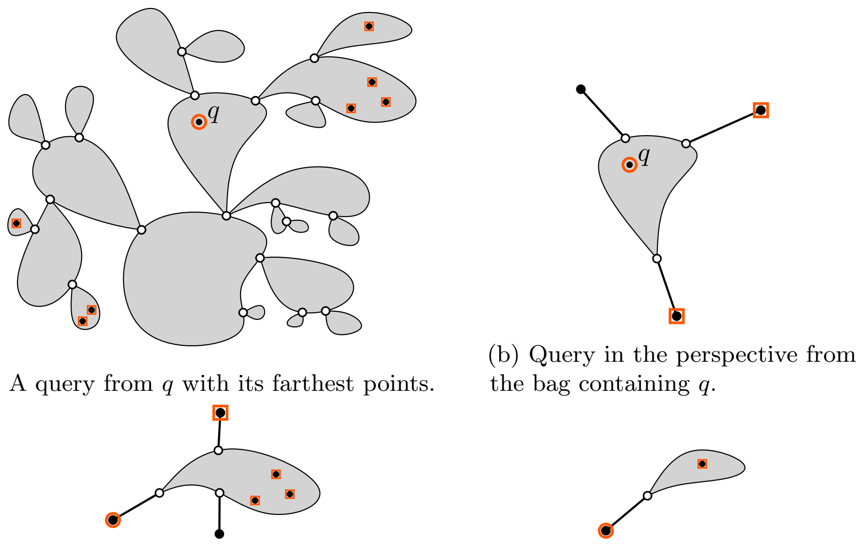
Finally, we obtain the missing value $\mathrm{ecc}_{\mathrm{bcut}(B',h^*)}(h^*)$ by taking the maximum in Equation (1). However, we need to avoid a dependence on the number of bags containing h^*, i.e., the degree of h^* in the tree structure. We augment the arcs of $T(G)$ with the following during the construction of B^*: When we compute the extended shortest path trees from the hinges of B^*, we store $\mathrm{ecc}_{\text{co-bcut}(B,h)}(h)$ with arc $h \to B$ in $T(G)$. These values are, at first, unknown for arcs in $T(G)$ on paths towards B^*. With each hinge vertex we store the largest and second largest known value among its adjacent arcs in $T(G)$ and two bags B_1 and B_2 attaining these values. How does this help us compute the maximum in Equation (1)? When we learn of the missing value $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ at h^* in $T(G)$, we have three cases depending on whether h^* has farthest points in the direction of B^* or, if not, in the direction of B'. First, $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ could be larger than $\mathrm{ecc}_{\text{co-bcut}(B_1,h^*)}(h^*)$ in which case $\mathrm{ecc}_{\mathrm{bcut}(B',h^*)}(h^*) = \mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$. Second, $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ could be strictly smaller than $\mathrm{ecc}_{\text{co-bcut}(B_1,h^*)}(h^*)$ with $B_1 \neq B'$ in which case $\mathrm{ecc}_{\mathrm{bcut}(B',h^*)}(h^*) = \mathrm{ecc}_{\text{co-bcut}(B_1,h^*)}(h^*)$. Third, $\mathrm{ecc}_{\text{co-bcut}(B^*,h^*)}(h^*)$ could be strictly smaller than $\mathrm{ecc}_{\text{co-bcut}(B_1,h^*)}(h^*)$ with $B_1 = B'$ in which case $\mathrm{ecc}_{\mathrm{bcut}(B',h^*)}(h^*) = \mathrm{ecc}_{\text{co-bcut}(B_2,h^*)}(h^*)$. With the aforementioned bookkeeping, we can handle each of the three cases in constant time and consequently construct $\mathrm{per}(B')$ in time proportional to the size of B'.

We construct $\mathrm{per}(B')$ re-using the construction of $\mathrm{per}(B^*)$ and the augmented tree structure. In the same way, we construct the perspectives from all other neighbors of B^*, then all perspectives from the neighbors of all neighbors of B^* and so forth for an overall construction time of $O(n)$. We inherit the query times from the data structures of trees and uni-cyclic networks; in summary this yields the following theorem for eccentricity queries in cactus networks.

Theorem 6 *Let G be a cactus network with n vertices. There is a data structure with $O(n)$ size and construction time supporting eccentricity queries on G in $O(1)$ time from branches and in $O(\log l)$ time from blocks of size l.*

3.2 Farthest-Point Queries

Consider a farthest-point query from a point q in bag B, as illustrated in Figure 19. First, we perform a farthest-point query from q in the farthest-point data structure of the perspective of G from B. This query yields all farthest-points from q inside B and it returns the vertex $\hat{h}$ representing the bag-cut $\mathrm{bcut}(B,h)$ when q has farthest points in $\mathrm{bcut}(B,h)$. Any path from q to a farthest point $\bar{q}$ in $\mathrm{bcut}(B,h)$ passes through h, hence $\bar{q}$ is also farthest from h in $\mathrm{bcut}(B,h)$. Recall that $\mathrm{bcut}(B,h)$ consists of the co-bag-cuts $\text{co-bcut}(B',h)$ of the bags B' neighboring B at h. To find all farthest points from q in $\mathrm{bcut}(B,h)$, we cascade the query into the co-bag-cuts $\text{co-bcut}(B',h)$ of those neighbors B' of B that contain farthest points from h in $\mathrm{bcut}(B,h)$, i.e., $\mathrm{ecc}_{\text{co-bcut}(B',h)}(h) = \mathrm{ecc}_{\mathrm{bcut}(B,h)}(h)$. For the decision into which co-bag-cuts to cascade, we consult the eccentricity values stored with the arcs of the tree structure.



(a) A query from q with its farthest points.

(b) Query in the perspective from the bag containing q.

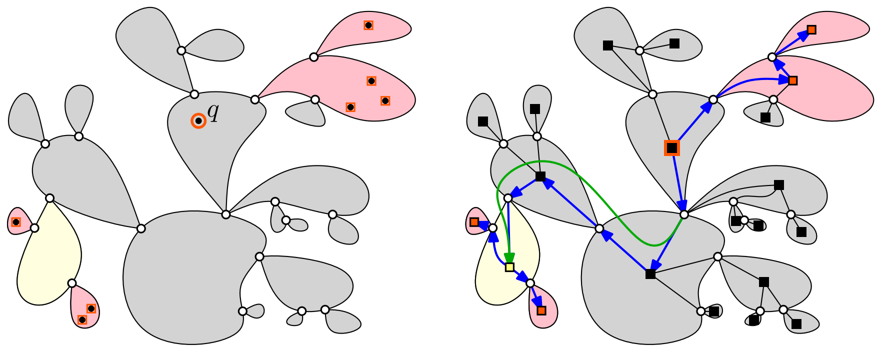
(c) Cascading the query in adjacent bags.

(d) Cascading the query further.

Figure 19: Answering a farthest-point query from q using the perspectives. Query points are indicated with orange circles, farthest-points with orange squares. To process the query, we first query for q in the perspective from the bag containing the query point (b). Then we cascade the query into adjacent bags that lead to further farthest points (c and d). We show only two cascaded queries, to report all farthest points, we need five more cascaded queries.

Assume we cascade a query from q into the co-bag-cut $\text{co-bcut}(B',h)$, as in Figures 19c and 19d. To query for the farthest points from h in $\text{co-bcut}(B',h)$, we perform a farthest-point query from $\hat{h}'$ in $\mathrm{per}(B')$, where $\hat{h}'$ is the vertex representing $\mathrm{bcut}(B',h)$ in $\mathrm{per}(B')$. The farthest points from $\hat{h}'$ in $\mathrm{per}(B')$ are also farthest from h in $\mathrm{per}(B') - \hat{h}'h$, which corresponds to $\text{co-bcut}(B',h)$. Since

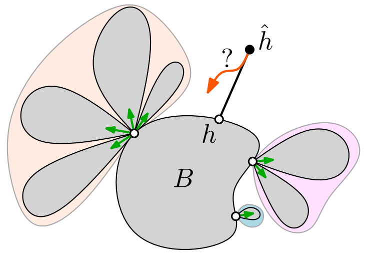
$\hat{h}'$ is a pendant vertex in $\mathrm{per}(B')$, this query takes time proportional to the number of farthest-points in $\mathrm{per}(B')$. The farthest points from $\hat{h}'$ in $\mathrm{per}(B')$ that lie in B' are farthest points form the original query point q, and those farthest points from $\hat{h}'$ in $\mathrm{per}(B')$ that represent bag-cuts adjacent to B' indicate where we have to continue our search for other farthest points from q. We propagate the query from q to the bags of G along paths to farthest points from q. However, we might visit $\Omega(n)$ bags before we reach one that contains a farthest point from q. We improve the query time by introducing shortcuts in the tree structure to bypass long chains of bags without farthest points.



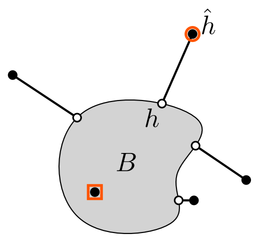
(a) A query from q with its farthest points. (b) Traversal of the tree structure.

Figure 20: The traversal of the tree strucuture (b) during a farthest-point query (a). The bags containing farthest-points are colored red, the bags where two paths to farthest points split are colored yellow, and all other blocks are colored gray. In blue, we highlight the arcs visited during the query and, in green, we highlight a shortcut bypassing several gray bags.

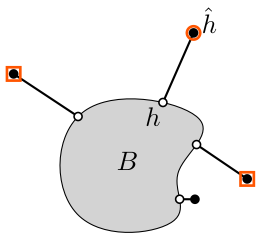
Consider a co-bag-cut $\text{co-bcut}(B, h)$ of bag B at hinge h. This co-bag-cut corresponds to the arc from h to B in the tree structure and all bags in $\text{co-bcut}(B, h)$ are in the sub-tree $T_{h\to B}$ reachable from h through B. To give a visual idea of our shortcuts, imagine we do the following: First, we color a bag B' in $T_{h\to B}$ red when B' contains farthest points from h in $\text{co-bcut}(B, h)$. Second, we color an uncolored bag B' in $T_{h\to B}$ orange when two paths from h to red bags split at B'. Finally, we color the remaining bags black. We seek to bypass black bags, since these are irrelevant for our query. For now, we only consider arcs of $T_{h\to B}$ leading away from h. The shortcut for arc $h' \to B'$ in $T_{h\to B}$ leads to the first arc $h'' \to B''$ where B'' is the closest orange or red descendant of h'. The shortcuts reachable from h form a directed tree where all vertices representing bags are either yellow or red; following all shortcuts in this tree takes time $O(r)$ where r is the number of red bag vertices in $T_{h\to B}$. Using these shortcuts, the number of bags visited when reporting farthest points from h in $\text{co-bcut}(B, h)$ is linear in the number of bags containing these farthest points. Figure 20 shows an example of a farthest-point query with and without shortcuts.



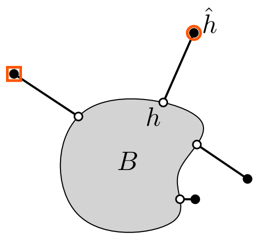
(a) The missing shortcut.



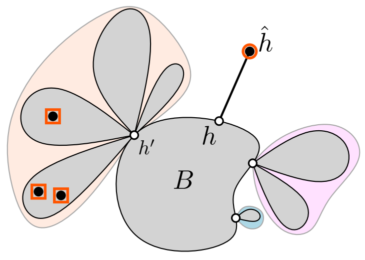
(b) We do not place a shortcut when the perspective from B contains any farthest points within B itself.



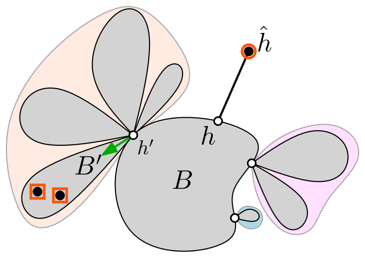
(c) We do not introduce a shortcut when there are farthest points in several adjacent bag-cuts.



(d) When only one bag-cut contains farthest-points, we take a closer look at its co-bag-cuts.



(e) We do not introduce a shortcut when several co-bag-cuts contain farthest points.



(f) We introduce a shortcut when only one co-bag-cut contains farthest points.

Figure 21: Determining a missing shortcut into a co-bag-cut of B at h, where the shortcuts for all co-bag cuts of the other hinges of B are known (a). We consider the farthest points from $\hat{h}$ in per(B): we place no shortcut when (b) there are farthest points in B, when (c) paths to farthest points split at B, and when (d,e) all farthest points lie in multiple co-bag-cuts of the same adjacent bag-cut. We only place a shortcut when (f) all farthest points lie in a single co-bag cut co-bcut(B', h'). In this case, the shortcut from co-bcut(B, h) leads to the target of the shortcut into co-bcut(B', h').

How can we determine these shortcuts efficiently? Recall that the propagation scheme starts at some bag B^* where we first compute the extended shortest path tree of each hinge h in B^* in $\mathrm{bcut}(B^*, h^*)$. We first consider only the arcs of $T(G)$ leading away from B^*. The arcs leading to leaf bags of $T(G)$ need no shortcuts. Consider an arc $h \to B$ and assume that all other arcs in the sub-tree $T_{h\to B}$ already have their shortcuts, as illustrated in Figure 21. We distinguish three cases using a query in $\mathrm{per}(B)$: First, there could be farthest points from $\hat{h}$ in B (red case). Second, there could be farthest points from $\hat{h}$ in two bag cuts of B (orange case). Third, all farthest points from $\hat{h}$ could lie in a single bag cut $\mathrm{bcut}(B, h')$ (black or orange case). In the first two cases, we introduce a trivial shortcut, i.e., the shortcut from $h \to B$ points to $h \to B$. In the third case, we inspect the information stored at the arcs incident to hinge h' to determine whether the farthest points from $\hat{h}$ in $\mathrm{bcut}(B, h')$ lie in multiple co-bag-cuts at h' (orange case) or in a single co-bag-cut $\text{co-bcut}(B', h')$ (black case). In the black case, we introduce a shortcut from $h \to B$ to the destination of the shortcut from $h' \to B'$. We obtain all shortcuts in the tree structure leading away from B^* without increasing our asymptotic bound on the size and construction time.

We employ our breadth-first-search propagation scheme to construct the shortcuts in the tree structure for arcs pointing towards B^*. At any hinge h of B^*, we are only missing the shortcut for the arc $h \to B^*$. With a query in $\mathrm{per}(B^*)$, we can immediately determine whether the red, the orange or the black case from above applies; no shortcuts towards B^* are required for this step. With all shortcuts of the arcs from the hinges of B^* in place, we can compute the shortcuts towards B^* for the hinges of all blocks neighboring B^*, then all hinges of the bags neighboring the neighbors of B^* and so forth.

Placing the shortcuts during the construction of our data structure for eccentricity queries takes only constant additional time and space per shortcut. How much time does a farthest-point query take? First, we have to perform a farthest-point query in the perspective from the bag containing the query point. Then, we follow shortcuts towards bags containing more farthest points. The farthest-point queries in the perspective of subsequent bags B' take time linear in the number of farthest-points in $\mathrm{per}(B')$, since we query from pendant vertices of $\mathrm{per}(B')$. Moreover, as all visited blocks are either orange or red, the number of visited blocks is linear in the number of red blocks, i.e., blocks containing farthest points. Altogether, this yields a query time of $O(k)$ from branches and $O(k + \log l)$ from blocks of size l.

Theorem 7 *Let G be a cactus network with n vertices. There is a data structure with $O(n)$ size and construction time supporting farthest-point queries on G in $O(k)$ time from branches and in $O(k + \log l)$ time from blocks of size l, where k is the number of farthest-points.*

4 Conclusions and Future Work

In previous work [3], we obtain a data structure with $O(m^2 \log n)$ construction time for any network with n vertices and m edges, and with optimal query times for eccentricity queries and farthest-point queries. In this work, we improve the construction time to $O(n)$ for certain classes of networks without sacrificing optimal query time. In future work, we aim to achieve $o(m^2 \log n)$ construction time for more general classes of networks such as planar networks, k-almost-trees [10], and series-parallel graphs [6].

Acknowledgements

We would like to thank the anonymous referees for their constructive and insightful comments. Jean-Lou De Carufel was supported by le Fonds de recherche du Québec – Nature et technologies (FRQNT). Carsten Grimm was supported by a fellowship from the German Academic Exchange Service (DAAD).

References

[1] F. Aurenhammer, R. Klein, and D.-T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, 2013. DOI: `10.1142/8685`.

[2] B. Ben-Moshe, B. Bhattacharya, Q. Shi, and A. Tamir. Efficient algorithms for center problems in cactus networks. *Theoretical Computer Science*, 378(3):237 –252, 2007. DOI: `10.1016/j.tcs.2007.02.033`.

[3] P. Bose, K. Dannies, J.-L. D. Carufel, C. Doell, C. Grimm, A. Maheshwari, S. Schirra, and M. Smid. Network Farthest-Point Diagrams. *Journal of Computational Geometry*, 4(1):182–211, 2013. URL: `http://jocg.org/v4n1p8/`.

[4] P. Bose, J.-L. D. Carufel, C. Grimm, A. Maheshwari, and M. Smid. Optimal Data Structures for Farthest-Point Queries in Cactus Networks. In *Proceedings of the 25th Canadian Conference on Computational Geometry.* CCCG 2013. (Waterloo, Ontario, Canada, Aug. 8–10, 2013), 2013. URL: `http://cccg.ca/proceedings/2013/papers/paper_23.pdf`.

[5] R. E. Burkard and J. Krarup. A linear algorithm for the pos/neg-weighted 1-median problem on a cactus. *Computing*, 60(3):193–215, 1998. DOI: `10.1007/BF02684332`.

[6] R. J. Duffin. Topology of Series-Parallel Networks. *Journal of Mathematical Analysis and Applications*, 10(2):303–318, 1965. DOI: `10.1016/0022-247X(65)90125-3`.

[7] M. Erwig. The Graph Voronoi Diagram with Applications. *Networks*, 36(3):156–163, 2000. DOI: `10.1002/1097-0037(200010)36:3<156::AID-NET2>3.0.CO;2-L`.

[8] T. Furuta, A. Suzuki, and K. Inakawa. The k-th Nearest Network Voronoi Diagram and its Application to Districting Problem of Ambulance Systems. Technical Report (0501). Nanzan University, 2005. URL: `http://birdie.ic.nanzan-u.ac.jp/MCENTER/pdf/wp0501.pdf`.

[9] C. Grimm. Eccentricity Diagrams. On Charting Farthest-Point Information in Networks. Diplomarbeit (Master's Thesis). Magdeburg: Otto-von-Guericke Universität Magdeburg, Mar. 2012.

[10] Y. Gurevich, L. J. Stockmeyer, and U. Vishkin. Solving NP-Hard Problems on Graphs that are almost Trees and an Application to Facility Location Problems. *Journal of the ACM*, 31(3):459–473, 1984. DOI: `10.1145/828.322439`.

[11] S. L. Hakimi. Optimum Locations of Switching Centers and the Absolute Centers and Medians of a Graph. *Operations Research*, 12(3):450–459, 1964. JSTOR: `168125`.

[12] S. L. Hakimi, M. Labbé, and E. Schmeichel. The Voronoi Partition of a Network and its Implications in Location Theory. *ORSA Journal on Computing*, 4(4):412–417, 1992. DOI: `10.1287/ijoc.4.4.412`.

[13] P. Hämäläinen. The absolute Center of a unicyclic Network. *Discrete Applied Mathematics*, 25(3):311–315, 1989. DOI: 10.1016/0166-218X(89)90009-7.

[14] G. Y. Handler. Minimax Location of a Facility in an undirected Tree Graph. *Transportation Science*, 7(3):287–293, 1973. JSTOR: 25767706.

[15] P. Hansen, M. Labbé, and B. Nicolas. The Continuous Center Set of a Network. *Discrete Applied Mathematics*, 30(2-3):181–195, 1991. DOI: 10.1016/0166-218X(91)90043-V.

[16] F. Harary and G. Prins. The Block-Cutpoint-Tree of a Graph. *Publicationes Mathematicae Debrecen*, 13:103–107, 1966.

[17] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16:372–378, 6, June 1973. DOI: 10.1145/362248.362272.

[18] O. Kariv and S. L. Hakimi. An Algorithmic Approach to Network Location Problems. I: The p-Centers. *SIAM Journal on Applied Mathematics*, 37(3):513–538, 1979. DOI: 10.1137/0137040.

[19] R. K. Kincaid. Exploiting Structure: Location Problems on Trees and Tree-like Graphs. In, *Foundations of Location Analysis*, pp. 315–334. Springer US, 2011. DOI: 10.1007/978-1-4419-7572-0_14.

[20] R. K. Kincaid and T. J. Lowe. Locating an absolute center on graphs that are almost trees. *European Journal of Operational Research*, 44(3):357–372, 1990. DOI: 10.1016/0377-2217(90)90247-9.

[21] M. Kolahdouzan and C. Shahabi. Voronoi-based K Nearest Neighbor Search for Spatial Network Databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. VLDB '04. (Toronto, Ontario, Canada, Aug. 31–Sept. 3, 2004). VLDB Endowment, 2004, pp. 840–851. URL: http://www.vldb.org/conf/2004/RS21P6.PDF.

[22] M. Labbé, D. Peeters, and J.-F. Thisse. Location on Networks. In, *Network Routing*. Vol. 8, in Handbooks in Operations Research and Management Science, part 7, pp. 551–624. Elsevier, 1995. DOI: 10.1016/S0927-0507(05)80111-2.

[23] Y.-F. Lan, Y.-L. Wang, and H. Suzuki. A linear-time Algorithm for solving the Center Problem on weighted Cactus Graphs. *Information Processing Letters*, 71(5–6):205–212, 1999. DOI: 10.1016/S0020-0190(99)00111-8.

[24] A. Okabe and K. Sugihara. *Spatial Analysis Along Networks: Statistical and Computational Methods*. Of *Statistics in Practice*. John Wiley & Sons, Ltd., 2012. DOI: 10.1002/9781119967101.

[25] A. Okabe and A. Suzuki. Locational Optimization Problems solved through Voronoi Diagrams. *European Journal of Operational Research*, 98(3):445–456, 1997. DOI: 10.1016/S0377-2217(97)80001-X.

[26] A. Okabe, T. Satoh, T. Furuta, A. Suzuki, and K. Okano. Generalized Network Voronoi Diagrams: Concepts, Computational Methods, and Applications. *International Journal of Geographical Information Science*, 22(9):965–994, 2008. DOI: 10.1080/13658810701587891.

[27] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Of *Probability and Statistics*. John Wiley & Sons, Ltd., 2nd edition ed., 2000. DOI: 10.1002/9780470317013.

[28] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query Processing in Spatial Network Databases. In *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB '03. (Berlin, Germany, Sept. 9–12, 2003). VLDB Endowment, 2003, pp. 802–813. URL: http://www.vldb.org/conf/2003/papers/S24P02.pdf.

[29] C. S. ReVelle and H. A. Eiselt. Location Analysis: A Synthesis and Survey. *European Journal of Operational Research*, 165(1):1–19, 2005. DOI: 10.1016/j.ejor.2003.11.032.

[30] Q. Shi. Efficient Algorithms for Network Center/Covering Location Optimization Problems. PhD thesis. Burnaby, British Columbia, Canada: School of Computing Science, Simon Fraser University, 2008. URL: http://summit.sfu.ca/item/8893.

[31] D. Taniar and W. Rahayu. A Taxonomy for Nearest Neighbour Queries in Spatial Databases. *Journal of Computer and System Sciences*, 79(7):1017–1039, 2013. DOI: 10.1016/j.jcss.2013.01.017.

[32] B. Ç. Tansel. Discrete Center Problems. In, *Foundations of Location Analysis*, in International Series in Operations Research & Management Science, pp. 79–106. Springer US, 2011. DOI: 10.1007/978-1-4419-7572-0_5.

[33] B. Ç. Tansel, R. L. Francis, and T. J. Lowe. Location on Networks: A Survey. Part I: The p-Center and p-Median Problems. *Management Science*, 29(4):482–497, 1983. DOI: 10.1287/mnsc.29.4.482.

[34] B. Ç. Tansel, R. L. Francis, and T. J. Lowe. Location on Networks: A Survey. Part II: Exploiting Tree Network Structure. *Management Science*, 29(4):498–511, 1983. DOI: 10.1287/mnsc.29.4.498.