Optimal Data Structures for Farthest-Point Queries in Cactus Networks[*]

Prosenjit Bose[†] Jean-Lou De Carufel[†] Carsten Grimm[†‡] Anil Maheshwari[†] Michiel Smid[†]

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.

1 Introduction

1.1 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 shown in Fig. 1, we measure the distance between points $p, q \in G$ in terms of 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 underlying network is understood.

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 study trees, cycles, uni-cyclic networks and cactus networks. A *uni-cyclic network* is a network with exactly one simple cycle and a *cactus network* is a network where no two simple cycles share an edge.

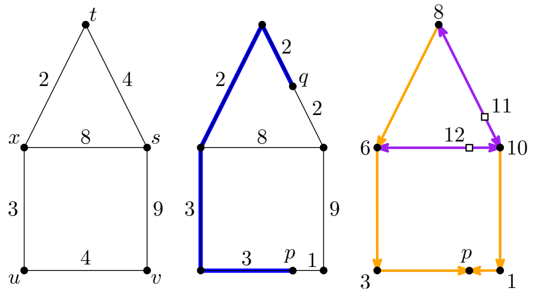

Figure 1: From left to right: (a) a network G (b) the network distance from $p = \frac{1}{4}u + \frac{3}{4}v$ to $q = \frac{1}{2}s + \frac{1}{2}t$ is $d_G(p,q) = 10$ (c) a shortest path tree rooted at p (orange[1]) and its extension (orange + purple). We have $\mathrm{ecc}(p) = 12$ and the farthest point from p is on xs.

1.2 Related Work

The problem of determining farthest points has been encountered [1, 2] when studying farthest-point Voronoi diagrams on networks. Specifically, when all of the infinitely many points on a network are considered sites. This point of view leads to a data structure with construction time $O(m^2 \log n)$ and size $O(m^2)$ supporting eccentricity queries and farthest point queries on arbitrary networks in optimal time [1, 2].

This work has connections to center problems [12, 11]. In a tree network, the set of farthest points changes only at its *absolute center* [4]. An *absolute center* is a point c on a network $G = (V,E)$ whose farthest vertices are as close as possible, i.e., $\max_{v \in V} d(c,v) = \min_{q \in G} \max_{v \in V} d(q,v)$. There are linear time algorithms for finding an absolute center in trees [6], uni-cyclic networks [5], and cactus networks [10]. The algorithm by Hämäläinen [5] plays an important role when we study uni-cyclic networks. We use the decomposition of a network into its tree structure like many works about center problems [9]. Tansel [12] and Kincaid [9] provide comprehensive surveys about center problems.

[*]Research supported in part by FQRNT and NSERC.

[†]School of Computer Science, Carleton University

[‡]Institut für Simulation und Graphik, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg

[1]Due to the limitations of the printed proceedings, please refer to the online version for colors in figures.

1.3 Our Contributions

We introduce optimal data structures supporting eccentricity queries and farthest-points queries for trees, cycles, uni-cyclic networks and cactus networks. The query times are summarized in Tab. 1. All of the presented data structures have linear construction time and, thus, require only linear space.

Type	Eccentricity	Farthest-Points
Tree	$O(1)$	$O(k)$
Cycle	$O(\log n)$	$O(\log n)$
Uni-Cyclic	$O(\log n)$	$O(k + \log n)$
Cactus	$O(\log n)$	$O(k + \log n)$

Table 1: The query times for different types of networks, where k is the number of reported farthest points.

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 query algorithms of Section 2 can be applied.

2 Trees, Cycles, and Uni-Cyclic Networks

2.1 Trees

Let T be a tree network. We call a point c on T whose farthest points are closest, a center of T, i.e, $\operatorname{ecc}(c) = \min_{x \in T} \operatorname{ecc}(x)$. A tree has exactly one center and we can find this center in linear time [6].

Lemma 1 *Let T be a tree, and let p be a point on T. All farthest points from p are leaves and any path from p to a farthest leaf contains the center of T.*

Corollary 2 *Let T be a tree with center c. For all points p on T we have $\operatorname{ecc}(p) = d(p, c) + \operatorname{ecc}(c)$.*

Splitting a tree T at its center c yields sub-trees with common farthest points, as shown in Fig. 2. When c is on edge uv with $u \neq c \neq v$, we split T into two sub-trees: the sub-tree T_u, containing the sub-edge uc, and the sub-tree T_v containing the sub-edge cv. The points on T_u (except for c) have all farthest points in T_v. The farthest points in c are those points that are farthest from T_u in T_v and those farthest from T_v in T_u.

Lemma 3 *Let T be a tree with center c, and let T' be one of the sub-trees obtained by splitting T at c. Leaf $l \in T'$ is farthest from $p \in T \setminus T'$ if and only if l is farthest from c, i.e., $\operatorname{ecc}_T(p) = d_T(l, p) \iff \operatorname{ecc}_T(c) = d_T(l, c)$.*

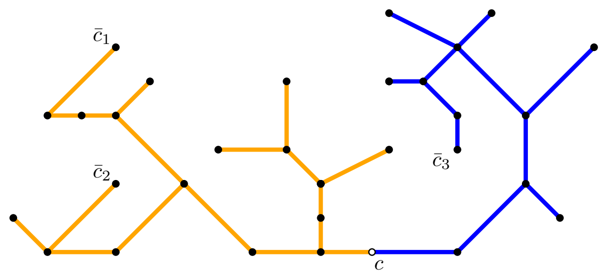

Figure 2: A tree T with geometric edge weights. The center c splits T into two sub-trees. For every point on the left sub-tree (orange) $\bar{c}_3$ is farthest and for every point on the right sub-tree (blue) $\bar{c}_1$ and $\bar{c}_2$ are farthest.

Using Corollary 2 and Lemma 3, we support eccentricity queries and farthest point queries in tree networks: Let T be a tree network with center c. We compute the position of c and the distances $d(c, v)$ for each vertex v of T. The maximum encountered distance is the eccentricity of c. Let $T_1, T_2, \ldots, T_r$ be the sub-trees obtained by splitting T at c. For each sub-tree, we store the set of farthest leaves from c in T_i, denoted by L_i, i.e., $L_i = \{l \in T_i \mid d(l, c) = \operatorname{ecc}(c)\}$. For an eccentricity query from point p on edge uv of T with $d(u, c) < d(v, c)$, we have $\operatorname{ecc}(p) = w_{up} + d(u, c) + \operatorname{ecc}(c)$. For a farthest-point query from p with $p \neq c$ and $p \in T_i$, we report all leaves in each L_j with $j \neq i$; for a farthest-point query from c we report the union of all L_i.

Theorem 4 *Let T be a tree network with n vertices. There is a data structure with construction time $O(n)$ supporting eccentricity queries on T in constant time and farthest-point queries on T in $O(k)$ time, where k is the number of reported farthest points.*

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 $\operatorname{ecc}(p) = d(p, \bar{p}) = w_C/2$. Supporting eccentricity queries on C amounts to calculating and storing $w_C/2$.

To support farthest-point queries, we compute the farthest-point $\bar{v}$ of each vertex v, subdivide the edge st containing $\bar{v}$ at $\bar{v}$, and introduce pointers between v and $\bar{v}$. We perform this computation as illustrated in Fig. 3: First, we compute the farthest point $\bar{v}$ for some initial vertex v by walking a distance of $w_C/2$ from v along C. Then, we sweep a point p from position $p = v$ to position $p = \bar{v}$ along C while maintaining the farthest point $\bar{p}$. During this sweep we subdivide C at p whenever $\bar{p}$ hits a vertex and at $\bar{p}$ whenever p hits a vertex. We store the distance from v to any other vertex, which enables us to compute the distance of any pair of vertices in constant time. The entire sweep takes linear time, thus, the resulting data structure occupies linear space.

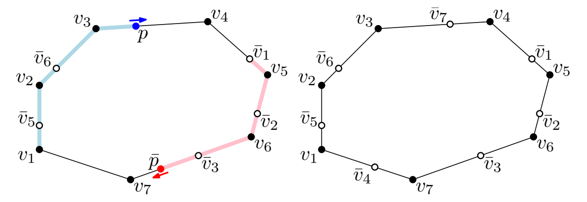

Figure 3: From left to right: (a) a sweep along cycle C starting from $p = v_1$ and (b) the resulting subdivision of C. The farthest point from any point on sub-edge $v_5\bar{v}_2$ is located on the sub-edge $\bar{v}_5 v_2$.

With the subdivided network, we can answer farthest-point queries in constant time, provided we know the sub-edge containing the query point p: When p is located on sub-edge $\bar{x}\bar{y}$ with $p = \mu\bar{x} + (1-\mu)\bar{y}$ for some $\mu \in [0,1]$ then $\bar{p}$ is located on xy with $\bar{p} = \mu x + (1-\mu)y$. The query point p is represented by the edge uv containing p and the value $\lambda \in [0,1]$ such that $p = \lambda u + (1-\lambda)v$. Using a binary search, we determine the sub-edge containing p and the value μ. This takes $O(\log n)$ time, since we subdivide each edge at most n times.

Lemma 5 *Given a cycle network C 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.*

2.3 Uni-cyclic Networks

As shown in Fig. 4, a *uni-cyclic* network U consists of a cycle C and trees $T_1, T_2, \ldots, T_r$, called the *branches*, attached to C at vertices $v_1, v_2, \ldots, v_r$, respectively.

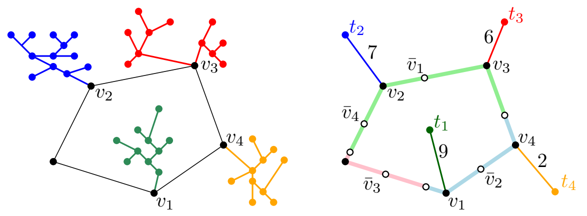

Figure 4: From left to right: (a) A unicyclic network with four branches (coloured) attached to its cycle. (b) The same network with compressed branches. The colouring of the cycle indicates the farthest branch.

Our data structure for uni-cyclic networks consists of three components: a data structure for queries on the cycle C that yields the farthest point among the points on C, a data structure for queries on C that yields the branches containing farthest points, and data structures for queries on the branches. The first component is the data structure from Section 2.2 supporting queries on C. The second component is a data structure supporting *farthest-branch queries*, i.e., queries for the branches containing farthest points from a query point on C. The second component uses the following simplification of U.

We replace each branch T_i of U with a vertex t_i and an edge $t_i v_i$, where v_i is the vertex connecting T_i to C. The weight of $t_i v_i$ is the farthest distance from v_i in T_i, i.e., $w_{t_i v_i} = \mathrm{ecc}_{T_i}(v_i)$. In the resulting network S, vertex t_i is farthest from p if and only if T_i contains farthest points from p with respect to U, i.e.,

$$d_S(p, t_i) = \mathrm{ecc}_S(p) \iff \exists q \in T_i \colon d_U(p,q) = \mathrm{ecc}_U(p) \ .$$

We call a vertex t_i *relevant* if there exists a point p on C who has t_i as a farthest vertex among $t_1, t_2, \ldots, t_r$, i.e., $d_S(t_i, p) = \max_{j=1}^{r} d_S(t_j, p)$. Recall that $\bar{v}_i$ denotes the farthest point from v_i among all points on C.

Lemma 6 *Vertex t_i is relevant if and only if t_i is farthest from $\bar{v}_i$ among $t_1, t_2, \ldots, t_r$.*

Lemma 6 yields a certificate for irrelevance. We say that t_j *dominates* t_i, and write $t_i \prec t_j$, if $d_S(t_i, \bar{v}_i) < d_S(t_j, \bar{v}_i)$. When t_j dominates t_i, all points on C are closer to t_i than to t_j and, thus, t_i cannot be relevant. Conversely, a vertex is relevant if and only if there is no other vertex dominating it. For the following, assume we have a circular list storing $t_1, t_2, \ldots, t_r$ ordered as $v_1, v_2, \ldots, v_r$ appear along the cycle.

Lemma 7 *Let t_a be the first relevant vertex after t_i and let t_b be the first relevant vertex before t_i. Vertex t_i is relevant if and only if neither t_a nor t_b dominate t_i, i.e., if and only if $t_i \not\prec t_a$ and $t_i \not\prec t_b$.*

Algorithm 1 computes the relevant vertices in $O(r)$ time using Lemma 7. We begin with a circular list containing all vertices $t_1, t_2, \ldots, t_r$. We remove irrelevant vertices from this list until no vertex in the list is dominated by its predecessor or successor. In each iteration of the while-loop we either delete some vertex or we mark the current t as processed ensuring that it will never assume the role of t again. Thus, the claim about the running time follows. Hämäläinen [5] uses a variant of Algorithm 1 in his linear time algorithm for finding the absolute center of a uni-cyclic network.

Algorithm 1: Determining the relevant vertices

```
input  : A circular list $L$ containing $t_1, t_2, \ldots, t_r$.
output: A sub-list of $L$ containing only the relevant
        vertices among $t_1, t_2, \ldots, t_r$.
Mark each $t_1, t_2, \ldots, t_r$ as unprocessed;
$t \leftarrow t_1$;
while $t$ is unprocessed do
    if $t \prec$ pred($t$) or $t \prec$ succ($t$) then
        $t \leftarrow$ succ($t$);
        delete(pred($t$));
    else if pred($t$) $\prec t$ then delete(pred($t$));
    else if succ($t$) $\prec t$ then delete(succ($t$));
    else ($t \not\prec$ pred($t$) $\not\prec t \not\prec$ succ($t$) $\not\prec t$)
        Mark $t$ as processed;
        $t \leftarrow$ succ($t$);
    end
end
```

The relevant vertices induce a subdivision of C into regions with a common farthest vertex among $t_1, t_2, \ldots, t_r$. When walking along C, we encounter these regions in the same order as the corresponding relevant points. Given the relevant vertices, we can compute the subdivision in linear time. Storing the relevant vertex with each sub-edge reduces a query for the farthest vertices among $t_1, t_2, \ldots, t_r$ to a binary search. We query for branches containing farthest points using the subdivision and our data structure for the cycle C.

Lemma 8 *Let U be a uni-cyclic network with n vertices and cycle C. There is a data structure with construction time $O(n)$ supporting farthest-branch queries in U from points on the cycle C in time $O(b + \log n)$, where b is the number of reported branches.*

Lemma 8 concludes the description of the second component of our data structure for uni-cyclic networks.

The third component is a data structure supporting queries on branches. Consider a branch T that is attached to the cycle C at vertex v. We extend T by a vertex v' and an edge vv' whose weight is the farthest distance from v to any point outside of T, i.e., $w_{vv'} = \mathrm{ecc}_{U \setminus T}(v)$. The resulting tree T', preserves farthest distances with respect to U, i.e., we have $\mathrm{ecc}_U(p) = \mathrm{ecc}_{T'}(p)$ for all $p \in T$. Thus, we can use the data structure from Section 2.1 to support eccentricity queries in U from points on T. Furthermore, if a point q outside of T has a farthest point $\bar{q}$ on T, then $\bar{q}$ is also farthest from v' in T'. When a farthest-branch query from q returns T, we report the farthest points from q in T with a farthest-point query from v' in T'.

A farthest point query in T' from a point $p \in T$ yields the farthest points from p on T and the vertex v' when p has farthest points outside of T. If v' is reported as a farthest point, we check whether $\bar{v}$, the farthest point from v on C is farthest from p. We determine the branches containing farthest points from p with a farthest-branch query at v and then report the farthest points from p in these branches as described above.

The above procedure for farthest point queries from T works correctly, unless the farthest branch query from v returns only T itself. This situation occurs for at most one branch of U, because it implies that all points on C have T as their only farthest branch. We resolve this issue by removing T from U and computing the farthest branches from v in the resulting network.

Theorem 9 *Let U be a uni-cyclic network with n vertices. There is a data structure with construction time $O(n)$ supporting eccentricity queries on U in $O(\log n)$ time and farthest-point queries on U in $O(k + \log n)$ time, where k is the number of reported 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 networks is a network where no two simple cycles share an edge. A *cut-vertex* is a vertex whose removal increases the number of connected components and a *bi-connected component* is a maximal connected sub-network without cut-vertices.

In linear time [8], we can decompose any network G into connected sub-networks $B_1, B_2, \ldots, B_b$ such that

- each edge of G is contained in exactly one B_i
- each B_i is a bi-connected component of G or the union of bi-connected components of G,
- and each vertex contained in more than one sub-network is a cut-vertex of G.

We call this a *block decomposition* of G into *blocks* $B_1, B_2, \ldots, B_b$. We call a cut-vertex contained in more than one block a *hinge vertex* [3]. For cactus networks, we consider the block decomposition where each block is a simple cycle or one of the (non-trivial) trees that remain when removing all cycles, as shown in Fig. 5.

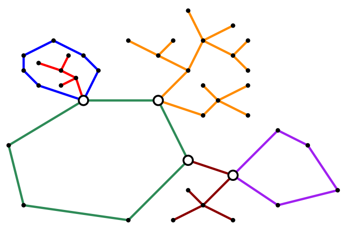

Figure 5: A cactus network decomposed into six blocks (coloured) with four hinge vertices (empty discs).

The following terms describe how we subdivide a network G with respect to a block decomposition; examples are shown in Fig. 6. For a sub-network S of G, we write $G - S$ to denote the network resulting from removing all edges of S from G (without removing any vertices). For a block B and a hinge vertex $h \in B$, we call the connected component of $G - B$ containing h the *block-cut* of B at h, denoted by $\mathrm{bcut}(B, h)$. We call the connected component of $G - \mathrm{bcut}(B, h)$ containing h the *co-block-cut* of B at h, denoted by $\mathrm{co\text{-}bcut}(B, h)$.

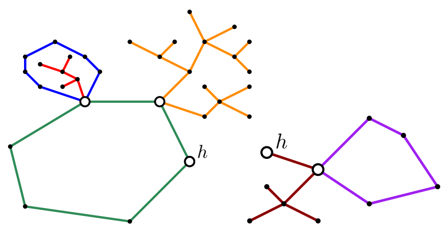

Figure 6: For the network from Fig. 5, from left to right: (a) the block-cut of the brown block at hinge vertex h and (b) the corresponding co-block-cut.

3.1 Eccentricity Queries

Consider a block B of a network G. To support eccentricity queries on B, we compress the (non-trivial) connected components of $G - B$ like we compress the branches of uni-cyclic networks. For each hinge vertex $h \in B$ 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 *locus* of B, denoted by $\mathrm{loc}(B)$. The locus of block B preserves farthest distances of G, i.e., $\mathrm{ecc}_{\mathrm{loc}(B)}(p) = \mathrm{ecc}_G(p)$ for all p on B.

We begin at some block B^* of a cactus network. For each hinge $h^* \in B^*$, we compute $\mathrm{ecc}_{\mathrm{bcut}(B^*,h^*)}(h^*)$ with a modified breadth-first-search in linear time. This breadth-first-search also yields the farthest distances along paths leading away from B^*, i.e., we obtain $\mathrm{ecc}_{\mathrm{bcut}(B,h)}(h)$ for any $\mathrm{bcut}(B, h) \subseteq \mathrm{bcut}(B^*, h^*)$.

Let B' be a block neighboring B^* at hinge vertex h, as shown in Fig. 7. To construct $\mathrm{loc}(B')$, we only lack the farthest distance from h in $\text{co-bcut}(B^*, h)$. We obtain this value with an eccentricity query in $\mathrm{loc}(B^*)$ via

$$\mathrm{ecc}_{\text{co-bcut}(B^*,h)}(h) = \mathrm{ecc}_{\mathrm{loc}(B^*)}(\hat{h}) - w_{h\hat{h}},$$

where $\hat{h}$ represents $\mathrm{bcut}(B^*, h)$ in $\mathrm{loc}(B^*)$. This way we obtain the loci of all neighbors of B^*, then all loci of the neighbors of all neighbors of B^* and so forth.

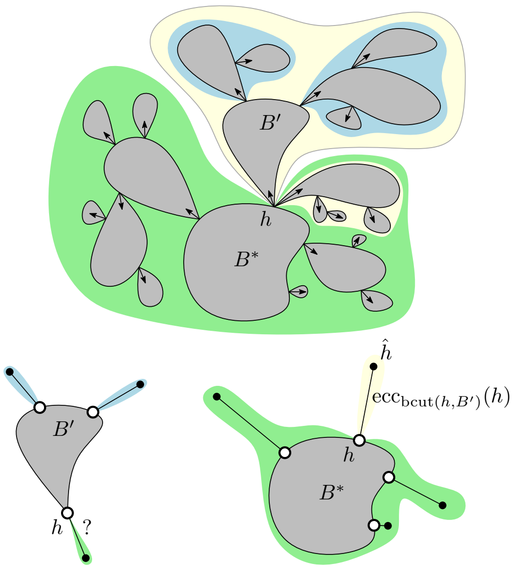

Figure 7: Top down and left to right: (a) An abstraction of the block structure of a network. The arrows indicate shortest path trees emanating from block B^*. (b) When constructing the locus of block B' we lack the distance from $\mathrm{bcut}(B', h)$ (green) whereas the distances from all other block cuts (blue) are known. (c) We obtain the missing distance with an eccentricity query in $\mathrm{loc}(B^*)$.

Constructing a data structure supporting eccentricity queries on a locus takes linear time in the size of the locus. Recall that each locus of a cactus network is either a tree or a uni-cyclic network. The eccentricity queries in a neighboring block take constant time, due to Theorem 9. Therefore, our data structure for eccentricity queries in cactus networks has construction time $O(n)$ and inherits the query times from uni-cyclic networks.

Lemma 10 *Let G be a cactus network with n vertices. There is a data structure with construction time $O(n)$ supporting eccentricity queries on G in $O(\log n)$ time.*

3.2 Farthest-Point Queries

To answer a farthest-point query from a point p in block B, we perform a farthest-point query in the locus $\mathrm{loc}(B)$ and then cascade the query into the neighboring blocks. If the query from p in $\mathrm{loc}(B)$ returns vertex $\hat{h}$ representing $\mathrm{bcut}(B, h)$, then $\mathrm{bcut}(B, h)$ contains farthest points from p. From the construction of $\mathrm{loc}(B)$, we know which blocks neighboring B at h lie on a path to from p to one of its farthest points. We continue with a farthest-point query from $\hat{h}$ in the loci of these blocks. This takes $O(n)$ time, since we might cascade through $O(n)$ blocks until we reach one containing farthest points from p. We improve the query time by using shortcuts to skip long chains of blocks without farthest points.

We define the *tree structure* [7] of a block decomposition of a network G, denoted by T_G, as the following graph. The set of vertices of T_G consists of the blocks of G and the hinge vertices of G. The edges of T_G connect a hinge vertex h and a block B whenever $h \in B$. Since the tree structure is indeed a tree [7] and since there are at most n blocks and at most n cut-vertices in a network with n vertices, T_G has at most $2n - 1$ edges.

The blocks and the hinge vertices visited during a cascading farthest-point query form a sub-tree T_{query} of the tree structure T_G. All farthest points from the query point are located in blocks that occur as vertices of T_{query}. Next, we use path compression to obtain a version of T_{query} whose size is linear in the number of blocks containing farthest points from the query point.

Consider an edge $\{h, B\}$ in T_G and the paths from h to blocks containing farthest points from h with respect to $\text{co-bcut}(B, h)$. We store a shortcut from $\{h, B\}$ to the first edge $\{h', B'\}$ along these paths, where B' contains farthest points or two paths split at B'. Fig. 8 shows a farthest-point query using one of these shortcuts. There are $O(n)$ shortcuts in total, since we add at most one shortcut per edge of T_G and since T_G has $O(n)$ edges.

We obtain the shortcuts leading away from B^* as a byproduct of the breadth-first-search used in the construction of the locus of B^*. For the remaining shortcuts, we rely on a similar strategy as used to obtain the loci of all blocks B with $B \neq B^*$. Let B be a block neighboring B^* at h. We introduce no shortcut

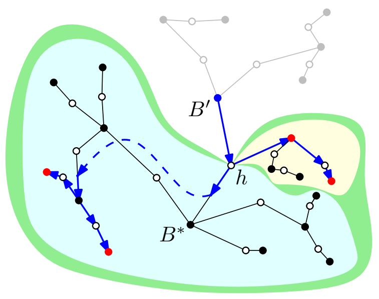

Figure 8: A farthest point query (blue) from block B' reporting the farthest points in $\text{bcut}(B', h)$ (green) using a shortcut (dashed). Blocks containing farthest points are indicated in red. An arc from a block B to a hinge vertex h indicates that we continue reporting farthest points in $\text{bcut}(B, h)$. An arc from h to B indicates that we continue reporting farthest points in $\text{co-bcut}(B, h)$.

when B^* contains farthest points from $\hat{h}$ or when two paths to farthest points from h in $\text{co-bcut}(B^*, h)$ split in B^* or at some hinge vertex of B^*. Otherwise, B^* has one neighboring block B' at hinge vertex h' such that $\text{co-bcut}(B', h')$ contains all farthest points from h in $\text{co-bcut}(B^*, h)$. In this case, we add a shortcut from $\text{co-bcut}(B^*, h)$ to the destination of the shortcut from $\text{co-bcut}(B', h')$. Since we conduct farthest point queries only on pendant edges of the loci, it takes constant time to determine which case applies and the overall construction time for cactus networks is $O(n)$.

Let p be a point in block B with k farthest points. During a farthest-point query from p, we report all farthest points from p in B and all block-cuts containing farthest points with a query in $\text{loc}(B)$. This takes $O(k + \log n)$ time due to Theorem 9. We follow the shortcuts associated to the reported block-cuts and obtain all other blocks containing farthest points in $O(k)$ time. For each reported block B' we perform a farthest-point query from a pendant vertex of $\text{loc}(B')$. By Theorem 4, this takes linear time in the number of farthest points in B'. The overall query time is $O(k + \log n)$.

Theorem 11 *Let G be a cactus network with n vertices. There is a data structure with construction time $O(n)$ supporting eccentricity queries on G in $O(\log n)$ time and farthest-point queries in $O(k + \log n)$ time, where k is the the number of reported farthest points.*

4 Conclusions and Future Work

In previous work [1, 2], we introduce a data structure with optimal query times for eccentricity and farthest-point queries and construction time $O(m^2 \log n)$ for any network with n vertices and m edges. In this work, we improve the construction time to $O(n)$ for certain classes of networks without sacrificing query time. In future work, we aim to achieve $o(m^2 \log n)$ construction time for more classes of networks.

References

[1] P. Bose, J.-L. D. Carufel, C. Grimm, A. Maheshwari, and M. Smid. On farthest-point information in networks. In *Proceedings of the 24th Canadian Conference on Computational Geometry*, pages 199–204, 2012.

[2] P. Bose, K. Dannies, J.-L. De Carufel, C. Doell, C. Grimm, A. Maheshwari, S. Schirra, and M. Smid. Network Farthest-Point Diagrams. *ArXiv*, Apr. 2013.

[3] 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.

[4] S. L. Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12(3):450–459, 1964.

[5] P. Hämäläinen. The absolute center of a unicyclic network. *Discrete Appl Math*, 25(3):311 – 315, 1989.

[6] G. Y. Handler. Minimax location of a facility in an undirected tree graph. *Trans. Sci.*, 7(3):287–293, 1973.

[7] F. Harary and G. Prins. The block-cutpoint-tree of a graph. *Publ. Math. Debrecen*, 13:103–107, 1966.

[8] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, 6 1973.

[9] R. K. Kincaid. Exploiting structure: Location problems on trees and treelike graphs. In *Foundations of Location Analysis*, pages 315–334. Springer US, 2011.

[10] Y.-F. Lan, Y.-L. Wang, and H. Suzuki. A linear-time algorithm for solving the center problem on weighted cactus graphs. *IPL*, 71(5–6):205–212, 1999.

[11] Q. Shi. *Efficient algorithms for network center/covering location optimization problems.* PhD thesis, School of Computing Science-Simon Fraser University, 2008.

[12] B. Ç. Tansel. Discrete center problems. In *Foundations of Location Analysis*, pages 79–106. Springer US, 2011.