Augmenting Graphs to Minimize the Radius

Joachim Gudmundsson ✉
The University of Sydney, Australia

Yuan Sha ✉
The University of Sydney, Australia

Fan Yao
The University of Sydney, Australia

Abstract

We study the problem of augmenting a metric graph by adding k edges while minimizing the radius of the augmented graph. We give a simple 3-approximation algorithm and show that there is no polynomial-time $(5/3 - \epsilon)$-approximation algorithm, for any $\epsilon > 0$, unless $P = NP$.

We also give two exact algorithms for the special case when the input graph is a tree, one of which is generalized to handle metric graphs with bounded treewidth.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases graph augmentation, radius, approximation algorithm

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.45

1 Introduction

We study the problem of minimizing the radius of a metric graph by inserting k new edges.

Let $G = (V, E)$ be a non-negative weighted graph with n vertices, and V^2 be the set of all possible edges on V. A non-edge of G, also referred to as shortcut, is an edge in $\bar{E} = V^2 \setminus E$. The graph distance $d_G(u, v)$ between two vertices $u, v \in V$ is the smallest weight of any path in G joining u and v. The eccentricity of a vertex $v \in V$ is the maximum graph distance between v and any vertex in V. The radius of G is the minimum eccentricity of any vertex in G and the center of G is a vertex with minimum eccentricity.

The radius of a graph is closely related to the diameter. The diameter of a graph is the maximum graph distance between any pair of vertices. The problem of minimizing the diameter of a graph by adding k new edges has been extensively studied [2, 6, 7, 8, 9, 14] and has applications in areas such as communication networks, information networks, flight scheduling and protein interaction.

In the general case each added edge may also have a cost, and the edge augmentation problem for minimizing the radius or diameter can then be seen as a bicriteria optimization problem. The two criteria are: (1) the total cost of the added edges, and (2) the radius or diameter of the augmented graph. A bicriteria optimization problem is then either (1) given a budget on the total cost of the added edges, minimize the radius or diameter, or (2) given a target on the radius or diameter of the augmented graph, minimize the total cost of the added edges. For radius, the first bicriteria optimization problem is formally defined as follows.

- PROBLEM: Bounded Cost Minimum Radius Edge Addition (BCMR)
- INPUT: An undirected graph $G = (V, E)$ with weight function $\ell : V^2 \to \mathbb{R}^+ \cup \{0\}$, a cost function $\varsigma : \bar{E} \to \mathbb{R}^+ \cup \{0\}$ and a positive number B.
- GOAL: Add a set $F \subseteq \bar{E}$ with $\sum_{e \in F} \varsigma(e) \leqslant B$ such that the radius of $\hat{G} = (V, E \cup F)$ is minimized.

© Joachim Gudmundsson, Yuan Sha, and Fan Yao;
licensed under Creative Commons License CC-BY 4.0
32nd International Symposium on Algorithms and Computation (ISAAC 2021).
Editors: Hee-Kap Ahn and Kunihiko Sadakane; Article No. 45; pp. 45:1–45:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we consider the BCMR problem restricted to the special case when the costs are unit and the weights satisfy the triangle inequality. That is, given a metric graph $G = (V, E, \ell)$, add $k = \lfloor B \rfloor$ shortcuts to G to minimize the radius of the augmented graph. We will refer to our restricted variant of the BCMR problem as the metric BCMR problem.

Related results

To the best of the authors' knowledge there is only one paper on the BCMR problem. Johnson and Wang [13] showed a linear-time algorithm for the special case when $k = 1$ and G is a path embedded in a metric space. They considered the continuous version where the center of the graph can be in the interior of an edge. However, the closely related problem of minimizing the diameter of the augmented graph has a long and rich history.

The problem of minimizing the diameter of the augmented graph such that the total cost of the shortcuts is within a budget (referred to as BCMD) and the problem of minimizing the cost of added shortcuts such that the diameter of the augmented graph is within a given value (referred to as BDMC) were shown to be NP-hard in [16] and $W[2]$-hard in [9, 11]. Li et al. [14] gave a constant factor approximation algorithm for the BCMD problem restricted to unit weights and unit costs. Later Bilò et al. [2] improved the analysis of Li et al.'s algorithm. Bilò et al.'s analysis also implies that Li et al.'s algorithm gives a 4-approximation for BCMD on metric graphs. Demaine and Zadimoghaddam [7] considered a variant of BCMD where the costs are unit and all non-edges have length δ, where δ is a small constant compared to the diameter of the graph. We will refer to this model as the DZ model. Demaine and Zadimoghaddam gave a constant factor approximation algorithm for the BCMD problem in the DZ model, which was later improved by Bilò et al. [2]. For the general BCMD problem where the weights and costs are arbitrary integers, Frati et al. [9] gave a 4-approximate Fixed-Parameter Tractable (FPT) algorithm (under the cost parameter). More restricted variants of the BCMD problem have also been considered in the literature, where the input graph is either a path or a tree and one shortcut is added. Section 1.1 in [13] gives a nice overview of these results.

Compared to the BCMD problem, the BDMC problem is traditionally harder to approximate. Dodis and Khanna [8] studied the BDMC problem in depth and gave extensive inapproximability results for different weight and cost functions. They also gave almost matching upper bounds for these variants of the BDMC problem.

Our results

In this paper we study the metric BCMR problem. Our main results are:

1. A simple $O(kn(m+n \log n))$ time 3-approximation algorithm and a $(5/3-\epsilon)$ approximation hardness bound, even for metric BCMR on geometric graph[1].
2. An exact $O(n^3 \log n)$ time algorithm for metric BCMR when the input graph is a tree and an exact $O((k^2b^2 + kb^3 2^b) \cdot n^{2b+2})$ time algorithm when the input graph has treewidth $b - 1$.

Our second result shows that the metric BCMR problem on trees is in P, while whether the BCMD problem on trees is in P is still an open problem, even for unit weights and unit costs.

[1] A geometric graph is a graph where all vertices are embedded in the Euclidean space.

Similar to Lemma 3 in [7], we can prove that any α-approximation for the BCMR problem is a 2α-approximation for the corresponding BCMD problem. Thus our algorithm for BCMR on metric graphs with bounded treewidth is a 2-approximation for BCMD on metric graphs with bounded treewidth, which is slightly better than Li et al.'s 4-approximation (however, their result holds for metric graphs).

Paper organization

The rest of the paper is organized as follows. The 3-approximation algorithm for the metric BCMR problem is presented in Section 2 and the approximation hardness results are given in Section 3. Section 4 describes our algorithms for the metric BCMR problem on trees. Finally, we present an algorithm on graphs with bounded treewidth in Section 5.

2 3-approximation algorithm

Let $G = (V, E, \ell)$ and k be an instance of the metric BCMR problem. Let F^* be an optimal solution for the instance, and let c^* be a center of $G^* = (V, E \cup F^*, \ell)$.

If the center c^* is known, then the metric BCMR problem is just adding k shortcuts to minimize the eccentricity of c^*. However, since c^* is not known, we will just try every vertex in V as a candidate. We define the Graph Augmentation Eccentricity Minimization problem, GAEM for short, as follows: given a metric graph $G = (V, E, \ell)$, an integer k and a vertex s in V, add k shortcuts in $\bar{E}$ to G such that the eccentricity of s is minimized. Obviously solving GAEM for every vertex in V gives us an optimal solution to the metric BCMR problem. Thus in the rest of this paper we will focus our attention on solving the GAEM problem.

The following lemma gives an important property for the GAEM problem, which we will utilize throughout the paper. The proof can be found in Appendix A.

▶ **Lemma 1.** *Given a metric graph $G = (V, E, \ell)$, an integer k and a vertex s in V for the GAEM problem, there is an optimal solution where every shortcut is incident to s.*

Lemma 1 allows us to only consider solutions in which all the shortcuts are incident to s. It also applies when we consider approximate solutions. Our 3-approximation algorithm uses the well-known farthest-first traversal technique popularized by Gonzalez [12]. Next we state the approximation algorithm.

Algorithm 1 FarthestAdditionGAEM(G, k, s).

Require: A metric graph $G = (V, E, \ell)$, an integer k and a vertex s in V.
Ensure: An approximate optimal solution for the GAEM instance.
1: $\hat{G} \leftarrow G$
2: **for** $i \leftarrow 1$ to k **do**
3: find the vertex u farthest from s in $\hat{G}$.
4: add (s, u) to $\hat{G}$.
5: **end for**
6: **return** $\hat{G}$

The approximation bound of Algorithm 1 is given in Lemma 2. The proof is found in Appendix B.

▶ **Lemma 2.** *Algorithm 1 is a 3-approximation algorithm for the GAEM problem.*

Algorithm 1 is very simple and can be implemented using standard graph algorithms. Let $n = |V|$ and $m = |E|$. In each iteration of the for loop, we use Dijkstra's algorithm to compute the single-source shortest paths from s to all other vertices in V. Using Fibonacci heaps [10], Dijkstra's algorithm runs in time $O(m' + n \log n)$, where $m' < m + k$. Without loss of generality, we may assume that $k \leqslant n - 1 \leqslant m$. The total running time is thus $O(km + kn \log n)$.

Running Algorithm 1 for n times gives our main result in this section.

▶ **Theorem 3.** *There is an $O(kn(m + n \log n))$ time, 3-approximation algorithm for the metric* BCMR *problem.*

When the input graph is unweighted or in the DZ model, arguments similar to the proof of Lemma 2 give an approximation factor of 2. For unweighted graphs, the 3-approximation algorithm runs in $O(knm)$ time by using BFS instead of Dijkstra's algorithm in Step 3 of Algorithm 1. Note that for general metric graphs we cannot use BFS in Step 3.

3 Approximation hardness

We show the approximation hardness of the metric BCMR problem by showing the approximation hardness of the GAEM problem. This follows from the fact that any polynomial time α-approximation algorithm for GAEM will trivially lead to a polynomial time α-approximation algorithm for the metric BCMR problem. The hardness proof is done using a reduction from the Dominating Set problem.

Given a graph $G = (V, E)$ and an integer k, the Dominating Set decision problem asks whether there is a subset $S \subset V$ of size k such that every vertex not in S is adjacent to a vertex in S. The GAEM decision problem is: given a metric graph $G = (V, E, \ell)$, a vertex $s \in V$, an integer k and a target value r, decide if there is a set F of k shortcuts such that their addition to G reduces the eccentricity of s to at most r.

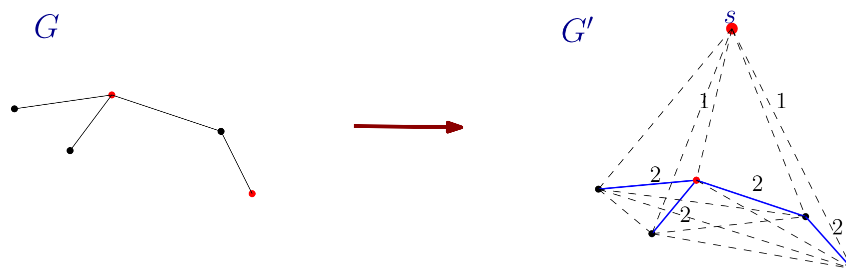

Figure 1 Reduction from the Dominating Set decision problem. Edges in E' are colored blue and have weight 2. All shortcuts are dashed and have weight 1.

▶ **Theorem 4.** *For any $\epsilon > 0$, finding a $(\frac{5}{3} - \epsilon)$ approximate solution for the* GAEM *problem is NP-hard.*

Proof. Let $G = (V, E)$ be a graph. Let $I = (G, k)$ be an instance of the Dominating Set decision problem. We transform I into an instance $I' = (G', s, k, 3)$ of the GAEM decision problem by adding an extra vertex s, that is, $G' = (V' = V \cup \{s\}, E' = E, \ell)$. We set the weight function ℓ as: the weight of any shortcut in $[V']^2 \setminus E'$ is 1 and the weight of any edge in E' is 2. See Figure 1. Note that ℓ satisfies triangle inequality.

If there is a dominating set $S \subset V$ of size k in G, we can choose the same k vertices in G' and add k shortcuts from s to every vertex in this subset. It follows that every vertex in V is connected to s by 1 shortcut and at most 1 edge in E. The eccentricity of s is thus at most 3.

If we can add k shortcuts to G' such that the eccentricity of s in the resulting graph is at most 3, we can add such k shortcuts which are all incident to s, by Lemma 1. For such k shortcuts, all other vertices of the shortcuts except s form a subset $S' \subset V$ of size k. Since the eccentricity of s is at most 3, every vertex in $V \setminus S'$ must be adjacent to a vertex in S' by an edge in E. Thus S' is dominating set of G.

Any vertex that is connected to s through 1 shortcut and 1 edge in E has distance 3 to s. Any vertex that is connected to s through 1 shortcut and 2 edges in E has distance 5 to s. If there is a $(\frac{5}{3} - \epsilon)$ approximation algorithm for the GAEM problem then we can use it to solve I' and thus any dominating set instance. Thus finding a $(\frac{5}{3} - \epsilon)$ approximate solution for the GAEM problem is NP-hard. ◀

The same construction can be used for the DZ model while only modifying the length function. Assuming the length of all non-edges to be very small, we get:

▶ **Lemma 5.** *For any $\epsilon > 0$, finding a $(2 - \epsilon)$ approximate solution for the GAEM problem in the DZ model is NP-hard.*

Note that this is a tight lower bound since Algorithm 1 is a 2-approximation algorithm for the GAEM problem in the DZ model.

Somewhat surprising is that the GAEM problem on geometric graphs (the weights are Euclidean distances) is as hard to approximate as the GAEM problem. This is stated in Lemma 6 and the proof is found in Appendix C.

▶ **Lemma 6.** *For any $\epsilon > 0$, finding a $(\frac{5}{3} - \epsilon)$ approximate solution for the GAEM problem on geometric graphs is NP-hard.*

4 The GAEM problem on metric trees

In this section we consider the GAEM problem on trees. We give two algorithms: (1) a near quadratic-time algorithm, and (2) a slower algorithm with running time $O(min\{k^2n^3, n^4\})$. The first algorithm is more efficient and elegant, however, we are not able to generalize it to graphs of bounded treewidth. Therefore we will describe the second algorithm below (Section 4.1) while the description of the first algorithm can be found in Appendix D. We state the result of the first algorithm here:

▶ **Theorem 7.** *The GAEM problem on trees can be solved in $O(n^2 \log n)$ time.*

Note that while Theorem 7 proves that the metric BCMR problem on trees can be solved in polynomial time, the complexity status for the BCMD problem on trees remains open, even for unit weights and unit costs.

Throughout this section we will assume that the input tree has degree at most 3. Otherwise, we transform it into a tree of degree 3 by splitting vertices with degree greater than 3. When a vertex of degree 3 is picked as the root, we split the vertex and get a binary tree. In the following, we assume that the input is a binary tree rooted at s.

4.1 The second algorithm for trees

By Lemma 1 all the shortcuts we add are incident to s. For a binary tree $T = (V, E, \ell)$ rooted at s, we aim to add k shortcuts all incident to s so that the eccentricity of s in the augmented graph is minimized.

Our algorithm is a dynamic programming approach with three parameters. Before we define the subproblems we need to define some notations.

For a vertex v in V, let $lc(v)$ and $rc(v)$ denote the left and right child of v, respectively. Let T_v denote the subtree of T rooted at v. Let $D^{\uparrow}(v) = \bigcup_{u \in T \setminus T_v} \{d_T(v,u) + |us|\}$ and let $D^{\downarrow}(v) = \bigcup_{u \in T_v} \{d_T(v,u) + |us|\}$. Each distance in $D^{\uparrow}(v)$ is the weight of a tree path from v to a vertex u in $T \setminus T_v$, plus the shortcut (u, s). The distances in $D^{\downarrow}(v)$ are defined in the same way but through a vertex u in T_v. This is illustrated in Fig. 2(a), where the weight of the blue path is in $D^{\uparrow}(v)$ and the weight of the red path is in $D^{\downarrow}(v)$. Note that the number of distances in $D^{\downarrow}(v)$ is $|T_v|$ and the number of distances in $D^{\uparrow}(v)$ is $|T \setminus T_v|$. In our recursion we will need the subset $D^{\downarrow}(v, d)$ of $D^{\downarrow}(v)$ containing all distances in $D^{\downarrow}(v)$ that are at least d.

For a subtree T_v and three parameters $d^{\uparrow}$, $d^{\downarrow}$ and $\bar{k}$, the subproblem is to find a set of $\bar{k}$ shortcuts from s to vertices in T_v so that the maximum distance from vertices in T_v to s is minimized. The parameter $d^{\uparrow}$ is the (assumed) weight of the shortest path from v to s via the parent of v (thus through a shortcut added to a vertex out of T_v), and the parameter $d^{\downarrow}$ is the (assumed) weight of the shortest path from v to s *not* going through the parent of v (thus through one of the $\bar{k}$ shortcuts). Note that $d^{\uparrow}$ is in $D^{\uparrow}(v)$ and its path goes through shortcut $(v^{\uparrow}, s)$ for some vertex $v^{\uparrow}$ in $T \setminus T_v$. Also $d^{\downarrow}$ is in $D^{\downarrow}(v)$ and its path goes through shortcut $(v^{\downarrow}, s)$ for some vertex $v^{\downarrow}$ in T_v.

For given $d^{\uparrow}$, $d^{\downarrow}$ and $\bar{k}$, let $R_v[d^{\uparrow}][d^{\downarrow}][\bar{k}]$ denote the minimum maximum distance from a vertex in T_v to s, when $\bar{k}$ shortcuts are allowed to be added from s to vertices in T_v.

We have two base cases. If $\bar{k} = 0$ then, by definition, $d^{\downarrow} = \infty$ and

$$R_v[d^{\uparrow}][d^{\downarrow}][0] = d^{\uparrow} + \max_{u \in T_v}\{d_T(u, v)\},$$

and if v is a leaf and $\bar{k} = 1$ then

$$R_v[d^{\uparrow}][d^{\downarrow}][1] = |vs|.$$

Next we state the recursion. Depending on the location of $v^{\downarrow}$, we have three different cases for $\bar{k} > 0$: (a) $v^{\downarrow}$ lies in $T_{lc(v)}$, (b) $v^{\downarrow}$ lies in $T_{rc(v)}$, or (c) $v = v^{\downarrow}$ which implies that there is a shortcut connecting s to v. In all three cases, the allowed $\bar{k}$ shortcuts are split between $T_{lc(v)}$ and $T_{rc(v)}$.

Case (a): $v^{\downarrow}$ lies in $T_{lc(v)}$.

$$\begin{aligned} R_v[d^{\uparrow}][d^{\downarrow}][\bar{k}] = \min_{1 \leqslant k' \leqslant \hat{k}} \{ \max\{ & R_{lc(v)}[d^{\uparrow} + |v, lc(v)|][d^{\downarrow} - |v, lc(v)|][k'], \\ & \min_{d \in D^{\downarrow}(rc(v), d^{\downarrow} - |v, rc(v)|)} \{R_{rc(v)}[\min\{d^{\uparrow}, d^{\downarrow}\} + |v, rc(v)|][d][\bar{k} - k']\}, \min\{d^{\uparrow}, d^{\downarrow}\}\}\}, \end{aligned}$$

where $\hat{k} = \min\{\bar{k}, |T_{lc(v)}|\}$. Note that k' shortcuts are added to vertices in $T_{lc(v)}$ and $\bar{k} - k'$ shortcuts are added to vertices in $T_{rc(v)}$. Since $v^{\downarrow}$ lies in $T_{lc(v)}$, the second parameter to $R_{lc(v)}$ equals $d^{\downarrow} - |v, lc(v)|$ and the second parameter to $R_{rc(v)}$ is at least $d^{\downarrow} - |v, rc(v)|$. See Figure 2(b). Finally, $\min\{d^{\uparrow}, d^{\downarrow}\}$ is the distance from v to s.

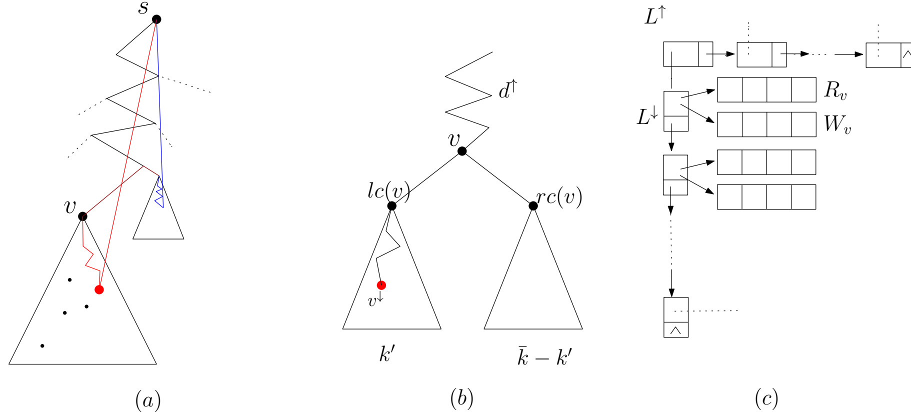

Figure 2 (a) Shortest paths from v to s. (b) $v^\downarrow$ lies in $T_{lc(v)}$. (c) Illustrating the list structures defined in Section 4.1.1.

Case (b): $v^\downarrow$ lies in $T_{rc(v)}$.

$$R_v[d^\uparrow][d^\downarrow][\bar{k}] = \min_{1\leqslant k'\leqslant \hat{k}}\{\max\{R_{rc(v)}[d^\uparrow + |v, rc(v)|][d^\downarrow - |v, rc(v)|][k'], \min_{d\in D^\downarrow(lc(v), d^\downarrow - |v,lc(v)|)}\{R_{lc(v)}[\min\{d^\uparrow, d^\downarrow\} + |v, lc(v)|][d][\bar{k} - k']\}, \min\{d^\uparrow, d^\downarrow\}\}\},$$

where $\hat{k} = \min\{\bar{k}, |T_{rc(v)}|\}$. The formula is symmetric to the formula for Case (a).

Case (c): $v = v^\downarrow$.

$$R_v[d^\uparrow][d^\downarrow][\bar{k}] = \min_{0\leqslant k'\leqslant \hat{k}}\{\max\{\min_{d_l\in D^\downarrow(lc(v),|s,lc(v)|)}\{R_{lc(v)}[|s, v| + |v, lc(v)|][d_l][k']\}, \min_{d_r\in D^\downarrow(rc(v)),|s,rc(v)|)}\{R_{rc(v)}[|s, v| + |v, rc(v)|][d_r][\bar{k} - k' - 1]\}, |vs|\}\},$$

where $\hat{k} = \min\{\bar{k} - 1, |T_{lc(v)}|\}$. The distance from v to s equals $|vs|$.

4.1.1 Data structures for DP

For efficient computation we define:

$$W_v[d^\uparrow][d^\downarrow][\bar{k}] = \min_{d\in D^\downarrow(v,d^\downarrow)}\{R_v[d^\uparrow][d][\bar{k}]\}.$$

Then the $\min_{d\in D^\downarrow(\dots)}$ terms in the formulae of R_v in the last subsection are replaced by W_v terms. By definition, W_v can be computed from R_v in an ordered manner.

For efficiency, we will use the following list structure for R_v and W_v in the dynamic programming steps. The first level is a list, $L^\uparrow$, ordered on the values of $d^\uparrow$. Each node in $L^\uparrow$ contains a second level list, denoted $L^\downarrow$, ordered on the values of $d^\downarrow$. And each node in $L^\downarrow$ contains two arrays of size at most k, one for R_v and one for W_v. See Fig. 2(c).

The dynamic programming is done in a bottom-up manner. At a leaf node v, $d^\uparrow$ has $n-1$ values and d^l_v has one value. We perform a tree traversal starting from v to get all the values of $d^\uparrow$, and order these values in $L^\uparrow$. For each $d^\uparrow$ value, build an $L^\downarrow$ list containing one node. At an internal node v, the values of its $L^\uparrow$ are obtained from the $L^\downarrow$ of its right child and the $L^\uparrow$ of its left child. Its $L^\downarrow$ is formed by merging the $L^\downarrow$ lists of its left and right child. The array elements of R_v are computed by using the formulae in the last subsection. Array elements of W_v are computed accumulatively from the array elements of R_v.

To get the shortcuts of an optimal solution, we need to keep some additional information. At a node v, the $L^{\uparrow}$ list also stores the vertex $v^{\uparrow}$ for each $d^{\uparrow}$ value. The $L^{\downarrow}$ list also stores the $v^{\downarrow}$ for each $d^{\downarrow}$ value. At an internal node v, when we use the formulae of R_v to compute an array element, we also keep track of the array elements of $lc(v)$'s structure and $rc(v)$'s structure that together give the solution for the current array element of R_v. When using the formula of W_v, we keep track of the array element of R_v that gives the value for the current array element of W_v. We can then backtrack with the above information to get the shortcuts of the optimal solution. The extra information we kept is constant per array element.

4.1.2 Running time

To analyze the running time, we first give two bounds. Their proofs are available in Appendix E. T is a rooted binary tree with n vertices.

▶ **Lemma 8.** $\sum_{v \in T} |T_{lc(v)}| \cdot |T_{rc(v)}| \leqslant n^2$.

▶ **Corollary 9.** $\sum_{v \in T} |T_v| \cdot \min\{|T_{lc(v)}|, |T_{rc(v)}|\} \leqslant 2n^2 + n \log n$.

At a leaf node v, the tree traversal for computing values of $d^{\uparrow}$ takes $O(n)$ time. Sorting these values takes $O(n \log n)$ time. Plus computing R_v and W_v, we spend $O(n \log n)$ time in total. At an internal node v, $O(n)$ time is spent on obtaining the sorted $d^{\uparrow}$ values for $L^{\uparrow}$ and the sorted $d^{\downarrow}$ values for $L^{\downarrow}$. $0 \leqslant k' \leqslant \min\{\bar{k}, |T_{vs}|\}$, where $|T_{vs}| = \min\{|T_{lc(v)}|, |T_{rc(v)}|\}$. $\bar{k} \leqslant \min\{k, |T_v|\}$. Thus computing the array elements of R_v for given $d^{\uparrow}$ and $d^{\downarrow}$ takes $O(\min\{k^2, |T_v| \cdot |T_{vs}|\})$ time. Both $d^{\uparrow}$ and $d^{\downarrow}$ have at most n values. Using $O(k^2)$ as the upper bound gives a total running time of $O(k^2 n^3)$. Using $O(|T_v| \cdot |T_{vs}|)$ as the upper bound gives a total running time $\sum_{v \in T} n^2 \cdot |T_v| \cdot |T_{vs}| = O(n^4)$, by Corollary 9. So the total running time is $O(min\{k^2 n^3, n^4\})$. To use less space, we can use *depth first search*. The space requirement is thus $O(k^2 n^2)$, and we get:

▶ **Theorem 10.** *When the input graph is a tree embedded in a metric space,* GAEM *can be solved in $O(min\{k^2 n^3, n^4\})$ time, using $O(k^2 n^2)$ storage.*

5 Metric BCMR on graphs with bounded treewidth

Treewidth, introduced by Robertson and Seymour [15], measures how similar a graph is to a tree. Many NP-hard graph problems can be solved efficiently when the input graph is restricted to graphs with bounded treewidth [1, 3]. The notion of treewidth is based on the notion of tree decomposition of a graph.

▶ **Definition 11** ([15]). *A tree decomposition of a graph $G = (V, E)$, denoted by $TD(G)$, is a pair (X, T) in which $T = (I, F)$ is a tree and $X = \{X_i | i \in I\}$ is a family of subsets of $V(G)$ such that:*
1. $\bigcup_{i \in I} X_i = V$;
2. *for each edge $e = \{u, v\} \in E$ there exists an $i \in I$ such that both u and v belong to X_i; and*
3. *for all $v \in V$, the set of nodes $\{i \in I | v \in X_i\}$ forms a connected subtree of T.*

X_i, a subset of $V(G)$, is called the *bag* of node i. The maximum size of a bag in $TD(G)$ minus one is called the *width* of the tree decomposition. The *treewidth* of a graph, denoted as $tw(G)$, is the minimum width over all possible tree decompositions of the graph. The

problem of deciding whether the treewidth of a given graph is at most k is NP-complete. However, there are efficient constructive algorithms when k is small ($k \leqslant 4$) or the input graph is outerplanar. There are also FPT approximation algorithms that guarantee a constant approximation factor [4].

A *nice* tree decomposition $(X = \{X_i | i \in I\}, T = (I, F))$ is a tree decomposition such that $|I| = O(tw(G) \cdot |V|)$, T is a rooted binary tree with three types of internal nodes:

1. a join node that has two children $lc(i), rc(i)$, $X_{lc(i)} = X_{rc(i)} = X_i$.
2. a forget node that has one child $lc(i)$, $X_i \subset X_{lc(i)}$ and $|X_i| = |X_{lc(i)}| - 1$.
3. an introduce node that has one child $lc(i)$, $X_{lc(i)} \subset X_i$ and $|X_{lc(i)}| = |X_i| - 1$.

Note that a tree decomposition can be transformed into a nice tree decomposition in linear time.

In this section, we assume that a nice tree decomposition (X, T) for G with width $tw(G)$ is given. We also assume that the bags of the root and the leaves are empty. Let T_i denote the subtree of T rooted at node i. Let $X_{T_i} = \bigcup\limits_{i \in T_i} X_i$ and $b = tw(G) + 1$.

We solve GAEM for graphs with bounded treewidth by generalizing the dynamic programming algorithm described in Section 4.1 for trees. Consider a tree $\hat{T} = (\hat{V}, \hat{E})$. A node $\hat{v}$ in $\hat{V}$ is the link point between the two subtrees $\hat{T}_{\hat{v}}$ and $\hat{T} \setminus (\hat{T}_{\hat{v}} \setminus \{\hat{v}\})$.

A tree decomposition defines a sequence of separators of the graph. The separator is the link between the two separated parts of the graph. For any $V' \subseteq V$, let $G(V')$ denote the subgraph of the input graph G induced by vertices in V'. For a node i of T in (X, T), X_i is a separator between $G(X_{T_i})$ and $G(V \setminus (X_{T_i} \setminus X_i))$. Each vertex in X_i is a link point. For each vertex $v_{i,j}$ in X_i, let $D^{\uparrow}(v_{i,j}) = \bigcup_{u \in V \setminus (X_{T_i} \setminus X_i)} (d_G(v_{i,j}, u) + |us|)$ and let $D^{\downarrow}(v_{i,j}) = \bigcup_{u \in X_{T_i}} (d_G(v_{i,j}, u) + |us|)$. Each distance in $D^{\uparrow}(v_{i,j})$ is the weight of a path from $v_{i,j}$ to a vertex u in $V \setminus (X_{T_i} \setminus X_i)$, plus the shortcut (u, s). We say the distance is *realized* by u. The distances in $D^{\downarrow}(v_{i,j})$ are defined in the same way but through a vertex u in X_{T_i}. As in Section 4.1 we will need the subset $D^{\downarrow}(v_{i,j}, d)$ of $D^{\downarrow}(v_{i,j})$, containing all distances in $D^{\downarrow}(v_{i,j})$ that are at least d.

At node i of (X, T), let $t_i^{\uparrow} = (d_{i,1}^{\uparrow}, \ldots, d_{i,|X_i|}^{\uparrow})$ denote a tuple where $d_{i,j}^{\uparrow} \in D^{\uparrow}(v_{i,j})$ is the weight of the shortest path from $v_{i,j}$ to s going through a shortcut added to a vertex in $V \setminus (X_{T_i} \setminus X_i)$, $1 \leqslant j \leqslant |X_i|$. Similarly we can define $t_i^{\downarrow} = (d_{i,1}^{\downarrow}, \ldots, d_{i,|X_i|}^{\downarrow})$. Let $v_{i,j}^{\uparrow}, 1 \leqslant j \leqslant |X_i|$ denote the vertex in $V \setminus (X_{T_i} \setminus X_i)$ that realizes $d_{i,j}^{\uparrow}$, and $v_{i,j}^{\downarrow}$ denote the vertex in X_{T_i} that realizes $d_{i,j}^{\downarrow}$. We say that $d_{i,j}^{\uparrow}$ and $d_{i,j}^{\downarrow}$ are *components* of $t_i^{\uparrow}$ and $t_i^{\downarrow}$, respectively. For a given node i and three parameters $t_i^{\uparrow}$, $t_i^{\downarrow}$ and $\bar{k}$ the subproblem is to find a set of $\bar{k}$ shortcuts from s to vertices in X_{T_i} so that the maximum distance from vertices in X_{T_i} to s is minimized, assuming the distance tuples $t_i^{\uparrow}$ and $t_i^{\downarrow}$.

Before we define the recursive steps we will need a few more notations. Let $R_i[t_i^{\uparrow}][t_i^{\downarrow}][\bar{k}]$ denote the minimum maximum distance from a vertex in X_{T_i} to s, where $\bar{k}$ shortcuts are allowed to be added between s and vertices in X_{T_i}, assuming the distance tuples $t_i^{\uparrow}$ and $t_i^{\downarrow}$.

In Section 4.1, we defined $W_{\hat{v}}$ for a link point $\hat{v}$ between $\hat{T}_{\hat{v}}$ and $\hat{T} \setminus (\hat{T}_{\hat{v}} \setminus \{\hat{v}\})$. For graphs of bounded treewidth we have $|X_i|$ link points at node i of (X, T). Assume i is a join node with left child $lc(i)$ and right child $rc(i)$. A component $d_{i,j}^{\downarrow}$ of $t_i^{\downarrow}$ is realized by a vertex $v_{i,j}^{\downarrow}$. If $v_{i,j}^{\downarrow}$ is in the left subtree $T_{lc(i)}$ of T_i then $d_{rc(i),j}^{\downarrow}$ must be at least $d_{i,j}^{\downarrow}$. The reverse is symmetrically true. So we let $t_i^{\downarrow} = (t_{i1}^{\downarrow}; t_{i2}^{\downarrow}) = \{d_{i,1}^{\downarrow}, \ldots, d_{i,|X_i|}^{\downarrow}\}$ be the $t^{\downarrow}$ parameter to a W_i. If $d_{i,j}^{\downarrow} \in t_{i1}^{\downarrow}$ then the component equals $d_{i,j}^{\downarrow}$. If $d_{i,j}^{\downarrow} \in t_{i2}^{\downarrow}$ then the component is at least $d_{i,j}^{\downarrow}$.

Thus

$$W_i[t_i^\uparrow][t_i^\downarrow = (t_{i1}^\downarrow; t_{i2}^\downarrow)][\bar{k}] = \min\{R_i[t_i^\uparrow][d_{i,1}, \ldots, d_{i,|X_i|}][\bar{k}]\},$$

where $d_{i,j} = d_{i,j}^\downarrow$ if $d_{i,j}^\downarrow \in t_{i1}^\downarrow$ and $d_{i,j} \in D^\downarrow(v_{i,j}, d_{i,j}^\downarrow)$ if $d_{i,j}^\downarrow \in t_{i2}^\downarrow$. There are $2^{|X_i|}-1$ nonempty subset of X_i, so we need to define $2^{|X_i|} - 1$ W_is with different components in $t_{i1}^\downarrow$ and $t_{i2}^\downarrow$.

Additional complexity for graphs with bounded treewidth. The definitions and notations above corresponds closely to similar concepts in Section 4.1. However, there are a couple of complications that we have to take care of specifically for graphs with bounded treewidth. Firstly, we have to handle the three types of nodes (join, introduce and forget nodes) differently. This will be discussed in detail below. Secondly, two aspects of the computation at a node of (X, T) become more complex.

1. *Feasible* values of $t_i^\uparrow$ and $t_i^\downarrow$. Not every combination of the values of each $d_{i,j}^\uparrow$ forms a feasible value of $t_i^\uparrow$. We say a $t_i^\uparrow$ is *feasible* only if it can be realized by some set of shortcuts (The formal definition appears in Appendix F).
2. There are $(2^{|X_i|} - 1)$ W_is to be computed from R_i.

We discuss how to handle these two problems in Appendix F.

Next we explain the recursion steps at join, introduce and forget nodes.

Join nodes. A join node i has two children $lc(i)$ and $rc(i)$. We need to compute R_i from $R_{lc(i)}$, $W_{lc(i)}$, $R_{rc(i)}$ and $W_{rc(i)}$. More specifically, we need to determine the tuples $t_{lc(i)}^\uparrow$, $t_{lc(i)}^\downarrow$, $t_{rc(i)}^\uparrow$ and $t_{rc(i)}^\downarrow$ from $t_i^\uparrow$ and $t_i^\downarrow$.

A partition of V for the separations between i and its children is shown in Figure 3(a). We use three copies of X_i for ease of illustration. For a component $d_{i,j}^\downarrow$ of $t_i^\downarrow$, $v_{i,j}^\downarrow$ can be a vertex in X_i, or in $(X_{T_{lc(i)}} \setminus X_i)$, or in $(X_{T_{rc(i)}} \setminus X_i)$. If $v_{i,j}^\downarrow$ is in X_i, both $d_{lc(i),j}^\downarrow$ and $d_{rc(i),j}^\downarrow$ equal $d_{i,j}^\downarrow$. Any path from $v_{i,j}$ to s that goes through a shortcut incident to a vertex in $X_{T_{rc(i)}} \setminus X_i$ has weight at least $d_{lc(i),j}^\downarrow$ so we can set $d_{lc(i),j}^\uparrow = d_{i,j}^\uparrow$, and similarly $d_{rc(i),j}^\uparrow = d_{i,j}^\uparrow$.

If $v_{i,j}^\downarrow$ is in $X_{T_{lc(i)}} \setminus X_i$, then $d_{lc(i),j}^\downarrow = d_{i,j}^\downarrow$ and $d_{rc(i),j}^\downarrow$ is at least $d_{i,j}^\downarrow$. Any path from $v_{i,j}$ to s that goes through a shortcut incident to a vertex in $X_{T_{rc(i)}} \setminus X_i$ has length at least $d_{lc(i),j}^\downarrow$ so we can set $d_{lc(i),j}^\uparrow = d_{i,j}^\uparrow$. For $rc(i)$, it is possible that $d_{i,j}^\downarrow < d_{i,j}^\uparrow$, so we set $d_{rc(i),j}^\uparrow = \min\{d_{i,j}^\uparrow, d_{i,j}^\downarrow\}$. The case when $v_{i,j}^\downarrow$ is in $X_{T_{rc(i)}} \setminus X_i$ is handled symmetrically.

The recursive relation for R_i is then:

$$R_i[t_i^\uparrow][t_i^\downarrow][\bar{k}] = \min_{k_1,k_2}\{\max\{W_{lc(i)}[t_{lc(i)}^\uparrow][(t_{lc(i)1}^\downarrow; t_{lc(i)2}^\downarrow)][k_1], \\ W_{rc(i)}[t_{rc(i)}^\uparrow][(t_{rc(i)1}^\downarrow; t_{rc(i)2}^\downarrow)][k_2]\}\},$$

where k_1 plus k_2 equals $\bar{k}$ plus the number of shortcuts added to vertices in X_i. The $t^\uparrow$ and $t^\downarrow$ tuples for $W_{lc(i)}$ and $W_{rc(i)}$ are derived as discussed above. Note that when $t_{lc(i)2}^\downarrow = \emptyset$, $W_{lc(i)} = R_{lc(i)}$.

Next we analyze the time spent at a join node. As shown in Appendix F, computing all feasible values of $t_i^\uparrow$ and $t_i^\downarrow$ for R_i requires $O(bn^b)$ time. For a given $t_i^\uparrow$ value, computing R_i for all feasible values of $t_i^\downarrow$ and $\bar{k}$ takes $O(k^2b \cdot |X_{T_i}|^{|X_i|}) = O(k^2bn^b)$ time. As discussed in Appendix F, computing all values for a W_i takes $O(kb^2n^{2b})$ time. Since there are $(2^b - 1)$ different W_is the total time spent at a join node is $O((k^2b + kb^22^b) \cdot n^{2b})$.

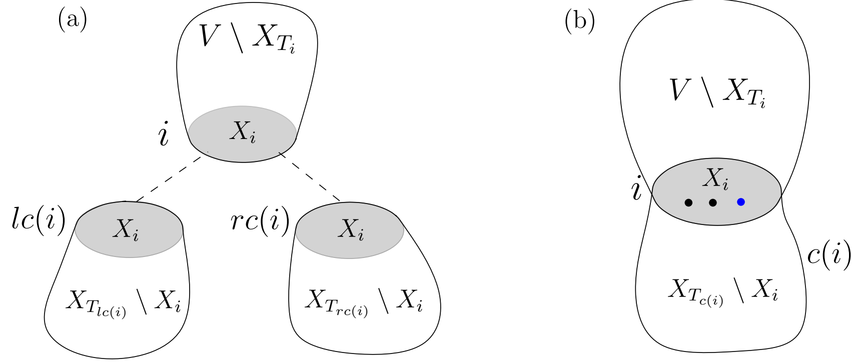

Figure 3 (a) Partition of V at a join node i for the separations between i and its children. (b) Separation at an introduce node i, $v_{i,|X_i|}$ is colored blue.

Introduce node. Consider an introduce node i and let $c(i)$ be its child. Without loss of generality, let $v_{i,|X_i|}$ be the vertex introduced at i. From the property of a tree decomposition, any path from $v_{i,|X_i|}$ to a vertex in $\{X_{T_i} \setminus X_i\}$ must go through a vertex in $X_i \setminus \{v_{|X_i|}\}$, see Fig. 3(b). We compute R_i from $R_{c(i)}$ or $W_{c(i)}$. For given $t_i^{\uparrow}$, $t_i^{\downarrow}$ and $\bar{k}$, we need to determine the corresponding $t_{c(i)}^{\uparrow}$ and $t_{c(i)}^{\downarrow}$. We consider two cases, depending on whether $v_{i,|X_i|}$ is incident to one of the $\bar{k}$ shortcuts, or not.

1. $d_{i,|X_i|}^{\downarrow} \neq |s, v_{i,|X_i|}|$, that is, $v_{i,|X_i|}$ is not incident to any of the $\bar{k}$ shortcuts. For any $d_{i,j}^{\uparrow}$, $j \neq |X_i|$, the corresponding $d_{c(i),j}^{\uparrow}$ is realized by the same vertex as $d_{i,j}^{\uparrow}$. Thus $d_{c(i),j}^{\uparrow} = d_{i,j}^{\uparrow}$, where $j \neq |X_i|$. Similarly, $d_{c(i),j}^{\downarrow} = d_{i,j}^{\downarrow}$, where $j \neq |X_i|$. For given $t_i^{\uparrow}$, $t_i^{\downarrow}$ and $\bar{k}$ at introduce node i, the maximum distance from all vertices in $X_{T_{c(i)}}$ to s is $R_{c(i)}[t_{c(i)}^{\uparrow}][t_{c(i)}^{\downarrow}][\bar{k}]$. The distance from $v_{i,|X_i|}$ to s is the minimum of $d_{i,|X_i|}^{\uparrow}$ and $d_{i,|X_i|}^{\downarrow}$. As a result we have:

$$R_i[t_i^{\uparrow}][t_i^{\downarrow}][\bar{k}] = \max\{R_{c(i)}[t_i^{\uparrow} \setminus d_{i,|X_i|}^{\uparrow}][t_i^{\downarrow} \setminus d_{i,|X_i|}^{\downarrow}][\bar{k}], \min\{d_{i,|X_i|}^{\uparrow}, d_{i,|X_i|}^{\downarrow}\}\}.$$

2. $d_{i,|X_i|}^{\downarrow} = |s, v_{i,|X_i|}|$, that is, $v_{i,|X_i|}$ is incident to one of the $\bar{k}$ shortcuts. By definition, $d_{c(i),j}^{\uparrow}$, $j \neq |X_i|$, equals the minimum of $d_{i,j}^{\uparrow}$ and $d_G(v_{i,j}, v_{i,|X_i|}) + |s, v_{i,|X_i|}|$. For any such $d_{i,j}^{\downarrow}, j \neq |X_i|$, that $v_{i,j}^{\downarrow}$ is not $v_{i,|X_i|}$, the corresponding $d_{c(i),j}^{\downarrow}$ equals $d_{i,j}^{\downarrow}$. For any such $d_{i,j}^{\downarrow}$, $j \neq |X_i|$, that $v_{i,j}^{\downarrow}$ is $v_{i,|X_i|}$, the corresponding $d_{c(i),j}^{\downarrow}$ is greater than $d_{i,j}^{\downarrow}$. For given $t_i^{\uparrow}$, $t_i^{\downarrow}$ and $\bar{k}$ at an introduce node i, the maximum distance from all vertices in $X_{T_{c(i)}}$ to s is $W_{c(i)}[t_{c(i)}^{\uparrow}][t_{c(i)}^{\downarrow}][\bar{k}-1]$. The distance from $v_{i,|X_i|}$ to s is $|s, v_{i,|X_i|}|$. Thus,

$$R_i[t_i^{\uparrow}][t_i^{\downarrow}][\bar{k}] = \max\{W_{c(i)}[(d_{c(i),1}^{\uparrow}, \ldots, d_{c(i),|X_i|-1}^{\uparrow})][(t_{c(i)1}^{\downarrow}; t_{c(i)2}^{\downarrow})][\bar{k}-1], |s, v_{i,|X_i|}|\},$$

where $d_{c(i),j}^{\uparrow} = \min\{d_{i,j}^{\uparrow}, d_G(v_{i,j}, v_{i,|X_i|}) + |s, v_{i,|X_i|}|\}$, for $j < |X_i|$. If $d_{i,j}^{\downarrow}$ is not realized by $v_{i,|X_i|}$, for $j < |X_i|$, then $d_{c(i),j}^{\downarrow} = d_{i,j}^{\downarrow}$ and is a component of $t_{c(i)1}^{\downarrow}$. Otherwise, $d_{c(i),j}^{\downarrow} > d_{i,j}^{\downarrow}$ and is a component of $t_{c(i)2}^{\downarrow}$.

The time required to compute the R_i values is $O(kbn^{2b})$. The W_is are computed from R_i in the same way as for a join node, hence, in totally $O(kb^2 2^b n^{2b})$ time. As a result, the time spent at an introduce node is $O(kb^2 2^b \cdot n^{2b})$.

Forget node. Assume i is a forget node and $c(i)$ is its child. Without loss of generality, assume $v_{c(i),|X_{c(i)}|}$ is the vertex forgotten at i. As usual, we compute R_i from values of $R_{lc(i)}$ or $W_{c(i)}$. $v_{i,j} = v_{c(i),j}, j < |X_{c(i)}|$. By definition, $d^{\uparrow}_{c(i),j} = d^{\uparrow}_{i,j}$, for all $j < |X_{c(i)}|$. In the input graph G, any path from $v_{c(i),|X_{c(i)}|}$ to a vertex in $V \setminus X_{T_i}$ must go through a vertex in X_i. Thus the corresponding $d^{\uparrow}_{c(i),|X_{c(i)}|}$ equals $\min\{d^{\uparrow}_{i,j} + d_G(v_{i,j}, v_{c(i),|X_{c(i)}|})|j < |X_{c(i)}|\}$. By definition, $d^{\downarrow}_{c(i),j} = d^{\downarrow}_{i,j}$ for all $j < |X_{c(i)}|$. However, $d^{\downarrow}_{c(i),|X_{c(i)}|}$ can take a number of different values. The only constraint is that it forms a feasible $t^{\downarrow}_{c(i)}$ together with all other $d^{\downarrow}_{c(i),j}$ values, $j < |X_{c(i)}|$. Thus R_i is computed from $W_{c(i)}$ where $d^{\downarrow}_{c(i),|X_{c(i)}|}$ is the only component in $t^{\downarrow}_{c(i)2}$ and takes the minimum value that forms a feasible $t^{\downarrow}_{c(i)}$ with all other $d^{\downarrow}_{c(i),j}$, $j < |X_{c(i)}|$.

$$R_i[t^{\uparrow}_i][t^{\downarrow}_i][\bar{k}] = W_{c(i)}[t^{\uparrow}_{c(i)}][t^{\downarrow}_{c(i)1} = t^{\downarrow}_i; t^{\downarrow}_{c(i)2}][\bar{k}],$$

where $d^{\downarrow}_{c(i),|X_{c(i)}|} \in t^{\downarrow}_{c(i)2}$. The W_is are computed from R_i as before. The time spent at a forget node is $O(kb^2 2^b \cdot n^{2b})$.

Since there are $O(bn)$ nodes in a nice tree decomposition we conclude this section with the following theorem.

▶ **Theorem 12.** *When the input graph G is a metric graph with bounded treewidth,* GAEM *problem can be solved in $O((k^2b^2 + kb^3 2^b) \cdot n^{2b+1})$ time, where $b = tw(G) + 1$. The metric* BCMR *problem on graphs with bounded treewidth can be solved in $O((k^2b^2 + kb^3 2^b) \cdot n^{2b+2})$ time.*

References

1. Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discret. Appl. Math.*, 23(1):11–24, 1989.
2. Davide Bilò, Luciano Gualà, and Guido Proietti. Improved approximability and non-approximability results for graph diameter decreasing problems. *Theor. Comput. Sci.*, 417:12–22, 2012.
3. Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 19–36, 1997.
4. Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016.
5. Gerard J. Chang and George L. Nemhauser. The k-domination and k-stability problems for sun-free chordal graphs. *SIAM J. Algebraic Discrete Methods*, 5(3):332–345, 1984.
6. Victor Chepoi and Yann Vaxès. Augmenting trees to meet biconnectivity and diameter constraints. *Algorithmica*, 33(2):243–262, 2002.
7. Erik D. Demaine and Morteza Zadimoghaddam. Minimizing the diameter of a network using shortcut edges. In *Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*, pages 420–431, 2010.
8. Yevgeniy Dodis and Sanjeev Khanna. Design networks with bounded pairwise distance. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 750–759, 1999.
9. Fabrizio Frati, Serge Gaspers, Joachim Gudmundsson, and Luke Mathieson. Augmenting graphs to minimize the diameter. *Algorithmica*, 72(4):995–1010, 2015.

10 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 338–346. IEEE Computer Society, 1984.

11 Yong Gao, Donovan R. Hare, and James Nastos. The parametric complexity of graph diameter augmentation. *Discret. Appl. Math.*, 161(10-11):1626–1631, 2013.

12 Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38:293–306, 1985.

13 C. Johnson and H. Wang. A linear-time algorithm for radius-optimally augmenting paths in a metric space. In *Proceedings of the 15th International Symposium on Algorithms and Data Structures*, pages 466–480, 2019.

14 Chung-Lun Li, S. Thomas McCormick, and David Simchi-Levi. On the minimum-cardinality-bounded-diameter and the bounded-cardinality-minimum-diameter edge addition problems. *Oper. Res. Lett.*, 11(5):303–308, 1992.

15 Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

16 Anneke A. Schoone, Hans L. Bodlaender, and Jan van Leeuwen. Diameter increase caused by edge deletion. *Journal of Graph Theory*, 11(3):409–427, 1987.

A Proof of Lemma 1

▶ **Lemma 1** *Given a metric graph $G = (V, E, \ell)$, an integer k and a vertex s in V for the* GAEM *problem, there is an optimal solution where every shortcut is incident to s.*

Proof. Let F^* be an optimal solution and $G^* = (V, E \cup F^*, \ell)$. Let $ecc^*(s)$ be the eccentricity of s in G^* and let T be the shortest path tree rooted at s. If every edge in F^* is incident to s then the lemma immediately holds. Otherwise there exists at least one edge (p, q) in F^* that is not incident to s. We will show that (p, q) can be replaced by either (s, p) or (s, q) such that the eccentricity of s in the resulting graph does not increase.

If (p, q) is an edge in T, then there is a subset U of V that go through (p, q) on their shortest paths to s in T. Without loss of generality assume $d_T(p, s) < d_T(q, s)$, as shown in Fig. 4. The set U is the vertices in the subtree T_q of T rooted at q. By triangle inequality $|sq| \le d_T(s, q)$. Replace (p, q) by (s, q) to get T'. Then every vertex in T'_q now has a path to s no longer than its shortest path in T, and every vertex not in T_q can still use their shortest path in T. Consequently, $ecc(s)$ in T' is at most $ecc^*(s)$.

If (p, q) is not an edge in T then replace (p, q) by either (s, p) or (s, q). Since every vertex can still use its shortest path in T the eccentricity of s does not increase. ◀

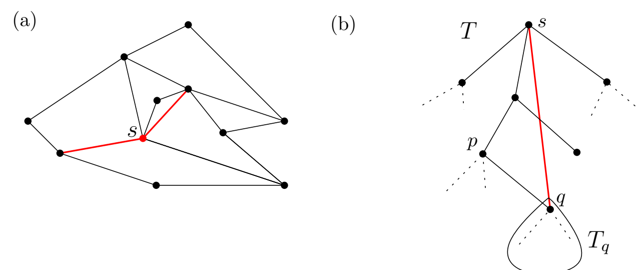

Figure 4 (a) An example where all the edges of the graph have unit length. The two shortcuts in red show an optimal solution for the GAEM instance with $k = 2$. (b) Illustrating the proof for Lemma 1.

B Proof of Lemma 2

▶ **Lemma 2** *Algorithm 1 is a 3-approximation algorithm for the* GAEM *problem.*

Proof. Let $A = \{q_1, q_2, \ldots, q_t\}$ be the set of vertices that are adjacent to s in G. If Algorithm 1 adds less than k shortcuts, the augmentation is optimal. Otherwise, let $U = \{p_1, p_2, \ldots, p_k\}$ be the set of vertices such that $(s, p_i)(i = 1, \ldots, k)$ is a shortcut added by Algorithm 1. Remember from Lemma 1 there is always an optimal solution F^* where all the shortcuts are incident to s. Assume $B = \{c_1, c_2, \ldots, c_k\}$ is the set of vertices such that (s, c_i) is a shortcut in F^*. Let $ecc^*(s)$ denote the eccentricity of s in $G^* = (V, E \cup F^*, \ell)$. $A \cup B$ is the set of vertices that are adjacent to s in G^*. Any vertex w other than s must go through exactly one vertex v in $A \cup B$ (may be itself) on its shortest path to s in G^*_s. Let $S(v), v \in A \cup B$ denote the set of vertices that go through v on their shortest paths to s in G^*. $\{S(v) : v \in A \cup B\}$ forms a partition of all the vertices other than s. We need to prove that the graph distance from w to s in $\hat{G}$ is at most $3ecc^*(s)$.

If $v \in A$, $d_G(w, s) \leqslant ecc^*(s)$ so the distance from w to s in $\hat{G}$ is at most $ecc^*(s)$. If any $p_i \in U$ is in a $S(q_j)$ where $q_j \in A$, $d_G(p_i, s) \leqslant ecc^*(s)$. At the time p_i is picked by Algorithm 1, p_i is the farthest vertex to s, so Algorithm 1 returns an optimal solution. What is left is when $v \in B$ and no $p_i \in U$ is in any $S(q_j)$ where $q_j \in A$. Any p_i must be in some $S(c_l)$ where $c_l \in B$.

If there is a $p_i \in U$ in $S(v)$, as in Figure 5(a), there is a path from w to s in $\hat{G}$ that goes through v and p_i and has weight $d_G(w, v) + d_G(v, p_i) + |sp_i|$. Both $d_G(w, v)$ and $d_G(v, u_i)$ are at most $ecc^*(s)$. By the triangle inequality, $|sp_i| \leqslant ecc^*(s)$. Thus the distance from w to s in $\hat{G}$ is at most $3ecc^*(s)$.

Otherwise, there is no $p_i \in U$ in $S(v)$. $|U| = |B|$. By the pigeonhole principle, there must exist a vertex $v' \in B$ such that two vertices $p_l, p_m \in U$ are in $S(v')$, as shown in Figure 5(b). Without loss of generality, assume p_l is picked before p_m in $\hat{G}$. When p_m is about to be added a shortcut, it is the farthest vertex to s. The distance from w to s in $\hat{G}$ is thus at most $d_G(p_m, v') + d_G(v', p_l) + |sp_l| \leqslant 3ecc^*(s)$, which is an upper bound of p_m's distance to s before it is added a shortcut. ◀

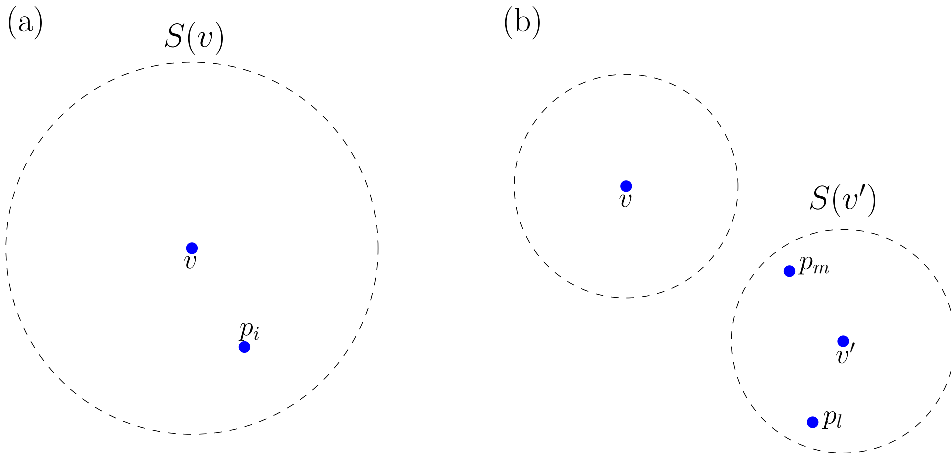

Figure 5 (a) There is a vertex $p_i \in U$ in $S(v)$. (b) There is no vertex $p_i \in U$ in $S(v)$.

C Proof of Lemma 6

▶ **Lemma 6** *For any $\epsilon > 0$, finding a $(\frac{5}{3} - \epsilon)$ approximate solution for the* GAEM *problem on geometric graphs is NP-hard.*

Proof. In [5], Chang and Nemhauser proved that the Dominating Set decision problem for bipartite graphs is NP-hard. We use a reduction from this problem to show the approximation hardness of GAEM on geometric graphs.

Let $G = (V = V_1 \cup V_2, E)$ be a bipartite graph where V_1 and V_2 are disjoint and every edge in E connects one vertex in V_1 to one vertex in V_2. Let $I = (G, k)$, $k < |V_1 \cup V_2|$, be an instance of the Dominating Set decision problem for bipartite graphs. Without loss of generality, assume $|V_1| \geqslant |V_2|$ and let $m = |V_1|$. We can embed G in the plane. Place vertices in V_1 at positions $(0,0)$,$(\epsilon/2(m-1), \epsilon/2(m-1))$,. . .,$(\epsilon/2, \epsilon/2)$ and place vertices in V_2 at positions $(2,0)$,$(2-\epsilon/2(m-1), 2-\epsilon/2(m-1))$,. . .,$(2-\epsilon(n-1)/2(m-1), 2-\epsilon(n-1)/2(m-1))$. Then add vertex s at position $(1,0)$. See Figure 6. We have a graph $G' = (V' = V \cup \{s\}, E' = E, \ell)$ where ℓ is the Euclidean distance function. I is transformed into an instance $I' = (G', s, k, 3)$ of GAEM on geometric graphs. For any point p in V_1 and any point q in V_2, we have $1 - \epsilon/2 \leqslant |sp| \leqslant 1$ and $2 - \epsilon \leqslant |pq| \leqslant 2$. G has a dominating set of size k if and only if we can add k shortcuts to G' so that the eccentricity of s in the resulting graph is at most 3. The shortest path from a vertex to s that goes through 1 shortcut and 2 edges in E has length at least $5 - 5\epsilon/2$. Since $3 \cdot (\frac{5}{3} - \epsilon) = 5 - 3\epsilon < 5 - 5\epsilon/2$, a $(\frac{5}{3} - \epsilon)$ approximation algorithm for GAEM on geometric graphs solves I' and thus any instance of the Dominating Set decision problem. ◀

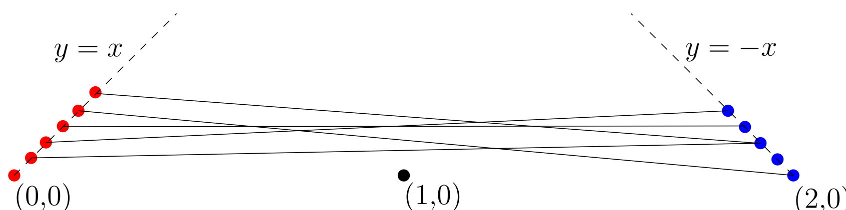

Figure 6 A bipartite graph embedded in the plane.

D An $O(n^2 \log n)$ time algorithm for GAEM on trees

We first devise an $O(\min\{k^2 n, n^2\})$ time algorithm that solves the decision problem of GAEM on trees. Then we search for an optimal solution by using the decision algorithm as a subroutine.

D.1 The decision algorithm

Given a value D, the decision algorithm decides whether we can add k shortcuts all incident to s to the inpdarkredut tree such that the eccentricity of s in the augmented graph is at most D. We introduce some notations that will be used in the discussion.

Let $d_T(u, v)$ denote the distance between any two vertices u and v in T. Let ζ be a set of $\bar{k}$ ($\bar{k} \leqslant k$) shortcuts added to vertices in T_v, where T_v is the subtree of T rooted at v. If from a vertex in T_v we can go along a path inside T_v and then through a shortcut in ζ to reach s within distance D, we say this vertex is covered by ζ, for example the red vertex

in Figure 7(a); otherwise we say it is uncovered by ζ. Let $U_v(\zeta)$ denote the set of vertices uncovered by ζ in T_v. When $U_v(\zeta)$ is empty, we say T_v is covered by ζ; otherwise we say T_v is uncovered by ζ. To reach s within D from an uncovered vertex in $U_v(\zeta)$, we have to go along a path out of T_v and then through a shortcut[2] added to a vertex in $T \setminus T_v$, if at all possible. See the red vertex p in Figure 7(b) for an example. Let $g_v(\zeta) = \max_{u \in U_v(\zeta)} d_T(u, v)$ denote the maximum distance from a vertex in $U_v(\zeta)$ to v in T. Let $d_v(\zeta)$ denote the length of the shortest path from v to s that goes through a shortcut in ζ.

The decision algorithm is used to solve an optimization problem. The optimization problem aims to find shortcuts that are optimal in incurring a "*Yes*" solution to the decision problem. Generally, given any integer $\bar{k} \leqslant k$ and any vertex v, the optimization problem is to add $\bar{k}$ shortcuts to vertices in T_v so that they are optimal in incurring a "*Yes*" solution to the decision problem on T. Before we formally define the meaning of being optimal in incurring a "*Yes*" solution, we first give a supporting lemma.

As discussed above, ζ is a set of $\bar{k}$ shortcuts added to vertices in T_v.

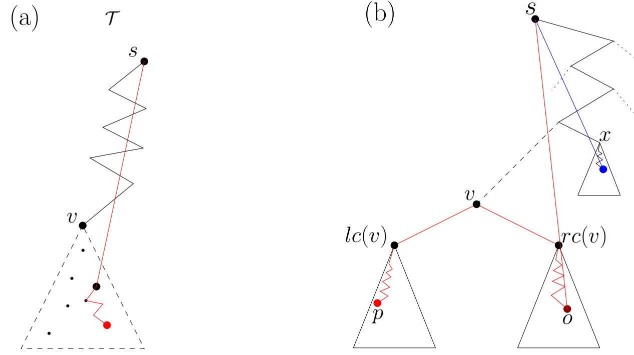

Figure 7 (a) $\bar{k}$ shortcuts are added to T_v. (b) Some vertices in T_v are uncovered.

▶ **Lemma 13.** *If $U_v(\zeta)$ is nonempty and ζ is part of a "Yes" solution to the decision problem of* GAEM *on T, then in the "Yes" solution*

1) *all vertices in $U_v(\zeta)$ go through the same shortcut added to a vertex in $T \setminus T_v$ on their shortest paths to s.*

2) *vertices in $T \setminus T_v$ does not go through any shortcut in ζ on their shortest paths to s.*

Proof. In the "*Yes*" solution, all vertices in $U_v(\zeta)$ must first go up to v, then follow the $v - s$ shortest path which cannot go through a shortcut in ζ. So property 1) is true. $U_v(\zeta)$ is nonempty, thus

$$g_v(\zeta) + d_v(\zeta) > D. \tag{1}$$

Let γ denote the shortcut that the shortest paths from vertices in $U_v(\zeta)$ to s go through. If γ is incident to a vertex in T_v's sibling T_w, for example the darkred vertex o in Figure 7(b), we have

$$g_v(\zeta) + |uv| + |uw| + d_w(\zeta') \leqslant D, \tag{2}$$

where ζ' is the set of shortcuts added to vertices in T_w. Equation 1 and 2 imply that $d_v(\zeta) > |uw| + d_w(\zeta')$, which means for every vertex in $T \setminus T_v$, the shortest path through γ to s is shorter than any shortest path through a shortcut in ζ. If γ is incident to a vertex in

[2] For ease of discussion, we consider the edges that are incident to s in T as pre-added shortcuts.

$T \setminus T_v \setminus T_w$, like the blue vertex in Figure 7(b), we can similarly show that for every vertex in $T \setminus T_v$, a shortest path through γ to s is shorter than any shortest path through a shortcut in ζ. Property 2) is proved. ◀

We now discuss the meaning of a ζ being optimal in incurring a "*Yes*" solution to the decision problem on T. Consider two sets ζ_1 and ζ_2:

1. if both $U_v(\zeta_1)$ and $U_v(\zeta_2)$ are nonempty. Without loss of generality, assume $g_v(\zeta_1) < g_v(\zeta_2)$. Assume ζ_2 is part of a "*Yes*" solution to the decision problem. Replace ζ_2 by ζ_1. Lemma 13 implies that the resulting solution is still a "*Yes*" solution. The reverse is not true. ζ_1 place lower requirement on shortcuts added to vertices outside T_v. Thus ζ_1 is better in making a "*Yes*" solution to the decision problem than ζ_2.
2. one of $U_v(\zeta_1)$ and $U_v(\zeta_2)$ is empty while the other is nonempty. Without loss of generality, assume $U_v(\zeta_1)$ is empty. Assume ζ_2 is part of a "*Yes*" solution to the decision problem. Replace ζ_2 by ζ_1. From Lemma 13(b), the resulting solution is still a "*Yes*" solution. ζ_1 places no requirement on the shortcuts added to vertices in $T \setminus T_v$ but ζ_2 does. Thus ζ_1 is better in making a "*Yes*" solution to the decision problem than ζ_2 .
3. both $U_v(\zeta_1)$ and $U_v(\zeta_2)$ are empty. Assume $d_v(\zeta_1) < d_v(\zeta_2)$. If ζ_2 is part of a "*Yes*" solution to the decision problem, replacing ζ_2 by ζ_1 will still give a "*Yes*" solution. But for vertices in $T \setminus T_v$, ζ_1 provides better shortcut than ζ_2. Thus ζ_1 is better in making a "*Yes*" solution to the decision problem.

We can formally define the above *better in making a 'Yes' solution* relation on ζs. Use $\prec$ to denote this relation. Then

$$U_v \text{ empty, smaller } d_v \prec U_v \text{ empty, greater } d_v \prec U_v \text{ nonempty, smaller } g_v \prec U_v \text{ nonempty, greater } g_v.$$

A ζ that has no other ζs preceding it in the $\prec$ relation is an optimal ζ.

The optimization problem is well-defined. By the optimal substructure of the optimization problem, we can use dynamic programming to solve the problem. Let $opt(v, \bar{k})$ denote an optimal solution of the problem on T_v with $\bar{k}$. When a solution uncovers T_v, we keep track of g_v. When a solution covers T_v, we keep track of d_v. The recursive formula of $opt(v, \bar{k})$ is:

$$opt(v, \bar{k}) = \min_{\prec}\{combine(f, opt(lc(v), k'), opt(rc(v), \bar{k} - k' - f)) | 0 \leqslant k' \leqslant \bar{k} - f, f = 0/1\},$$

where $\min_{\prec}$ is the "*minimum*" in relation "$\prec$" and f flags whether a shortcut is added to v. When combining the optimal sub-solutions of the left child $lc(v)$ and the right child $rc(v)$,

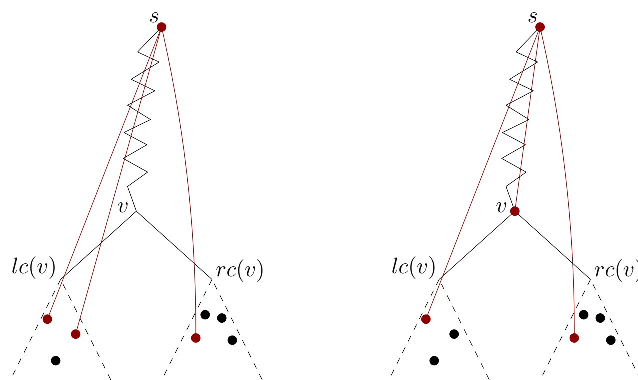

Figure 8 (a) (s, v) is not added as a shortcut. (b) (s, v) is added as a shortcut.

there are two cases: 1) v is not added a shortcut, 2) v is added a shortcut. For case 1), k' shortcuts are added to vertices in $T_{lc(v)}$ and $\bar{k}-k'$ shortcuts are added to vertices in $T_{rc(v)}$. For case 2), k' shortcuts are added to vertices in $T_{lc(v)}$ and $\bar{k}-1-k'$ shortcuts are added to vertices in $T_{rc(v)}$. See Figure 8 for an illustration. The *combine* procedures for case 1) and case 2) are similar and we only discuss case 1) below. $combine(0, opt(lc(v), k'), opt(rc(v), \bar{k} - k'))$ works as follows:

Case 1: $opt(lc(v), k')$ covers $T_{lc(v)}$ and $opt(rc(v), \bar{k}-k')$ covers $T_{rc(v)}$.Check if $\min\{|v, lc(v)| + d_{lc(v)}, |v, rc(v)| + d_{rc(v)}\} \leqslant D$. If so, v and all other vertices in T_v are covered and $d_v = \min\{|v, lc(v)| + d_{lc(v)}, |v, rc(v)| + d_{rc(v)}\}$; else T_v is uncovered and $g_v = 0$.

Case 2: $opt(lc(v), k')$ covers $T_{lc(v)}$ and $opt(rc(v), \bar{k} - k')$ uncovers $T_{rc(v)}$.We need to check whether vertices in $T_{rc(v)}$ uncovered by $opt(rc(v), \bar{k}-k')$ is covered by $opt(lc(v), k')$. Check if $g_{rc(v)} + |v, rc(v)| + |v, lc(v)| + d_{lc(v)} \leqslant D$. If so, T_v is covered and $d_v = |v, lc(v)| + d_{lc(v)}$; else T_v is uncovered and $g_v = g_{rc(v)} + |v, rc(v)|$.

Case 3: $opt(lc(v), k')$ uncovers $T_{lc(v)}$ and $opt(rc(v), \bar{k} - k')$ covers $T_{rc(v)}$.Symmetric to case 2.

Case 4: $opt(lc(v), k')$ uncovers $T_{lc(v)}$ and $opt(rc(v), \bar{k} - k')$ uncovers $T_{rc(v)}$.T_v is uncovered and $d_v = \max\{g_{lc(v)} + |v, lc(v)|, g_{rc(v)} + |v, rc(v)|\}$.

It is not hard to verify the correctness of the procedure. $combine(1, opt(lc(v), k')$, $opt(rc(v), \bar{k} - f - k'))$ works similarly and is left to the interested reader. The combining procedure at the root is slightly different at the root, we omit the details here.

All that is left is analyzing the running time of the dynamic programming. Once $opt(lc(v), k')$ and $opt(rc(v), \bar{k} - f - k')$ are known, the *combine* procedure takes constant time. Thus we only need to count the number of times $combine(0, \ldots)$ and $combine(1, \ldots)$ are called for computing every $opt(v, \bar{k})$. $\bar{k} \leqslant \min\{k, |T_v|\}$ and $k' \leqslant \min\{\bar{k}, |T_{lc(v)}\}$. Using $\bar{k} \leqslant k$ and $k' \leqslant \bar{k}$ as upper bounds, at a vertex v, $combine(0, \ldots)$ is called $O(\sum_{\bar{k}=1}^{k} \bar{k}) = O(k^2)$ times. Similarly, $combine(0, \ldots)$ is called $O(k^2)$ times. So we spend $O(k^2)$ time at a vertex v. This gives a total running time of $O(k^2 n)$. We also use $\bar{k} \leqslant |T_v|$ and $k' \leqslant |T_{lc(v)}|$ as upper bounds. We assume that $T_{lc(v)} = T_{vs}$, otherwise we can just swap $lc(v)$ and $rc(v)$. Thus for all vertices in T, $combine(0, \ldots)$ is called $O(\sum_{v \in T} |T_v| \cdot |T_{vs}|) = O(n^2)$ times, by Lemma 9. Similarly, for all vertices, $combine(1, \ldots)$ is called $O(n^2)$ times. This gives a total running time of $O(n^2)$. Thus the total running time is $O(\min\{k^2 n, n^2\})$.

▶ **Theorem 14.** *The decision problem of* GAEM *on trees can be solved in $O(min\{k^2 n, n^2\})$ time.*

D.2 The search algorithm

For any solution of k shortcuts all incident to s, there is a farthest vertex v to s in the augmented graph. The shortest path from s to v goes from s to a vertex u (which may be v) through a shortcut or an edge in T, then follows the shortest path from u to v in T. Both u and v are vertices in T. So there are at most n^2 possible values for the eccentricity of s in any augmented graph.

▶ **Lemma 15.** *There are at most n^2 possible values for the eccentricity of s in any augmented graph where all shortcuts are incident to s.*

Since our input is a tree, we can compute all possible values in $O(n^2)$ time. Then we sort the values and do a binary search over the sorted values, using the decision algorithm as a subroutine. The minimum value for which the decision algorithm returns a "*Yes*" solution is the minimum eccentricity incurred by an optimal solution. We can find an optimal solution in $O(\min\{k^2 n, n^2\} \cdot \log n + n^2 \log n) = O(n^2 \log n)$ time.

▶ **Theorem 16.** *GAEM on trees can be solved in $O(n^2 \log n)$ time.*

E Proofs of Lemma 8 and Corollary 9

▶ **Lemma 8** $\sum_{v \in T} |T_{lc(v)}| \cdot |T_{rc(v)}| \leqslant n^2$.

Proof. We use induction on the number of tree nodes to prove the bound. When $n = 1$, the tree contains only one node and the bound holds trivially.

Assume $\sum_{v \in T} |T_{lc(v)}| \cdot |T_{rc(v)}| \leqslant n^2$ holds for all $n < k$. Then for trees of size $n = k$, $\sum_{v \in T} |T_{lc(v)}| \cdot |T_{rc(v)}| = \sum_{v \in T_{lc(s)}} |T_{lc(v)}| \cdot |T_{rc(v)}| + \sum_{v \in T_{rc(s)}} |T_{lc(v)}| \cdot |T_{rc(v)}| + |T_{lc(s)}| \cdot |T_{rc(s)}|$, where s is the root of T. By the induction hypothesis, $\sum_{v \in T_{lc(s)}} |T_{lc(v)}| \cdot |T_{rc(v)}| \leqslant |T_{lc(s)}|^2$, $\sum_{v \in T_{rc(s)}} |T_{lc(v)}| \cdot |T_{rc(v)}| \leqslant |T_{rc(s)}|^2$. Thus $\sum_{v \in T} |T_{lc(v)}| \cdot |T_{rc(v)}| \leqslant |T_{lc(s)}|^2 + |T_{rc(s)}|^2 + |T_{lc(s)}| \cdot |T_{rc(s)}| \leqslant (|T_{lc(s)}| + |T_{rc(s)}|)^2 < n^2$. ◀

▶ **Corollary 9** $\sum_{v \in T} |T_v| \cdot \min\{|T_{lc(v)}|, |T_{rc(v)}|\} \leqslant 2n^2 + n \log n$.

Proof. Let T_{vs} be the subtree rooted at a child of v such that $|T_{vs}| = \min\{|T_{lc(v)}|, |T_{rc(v)}|\}$. We know that $\sum_{v \in T} |T_v| \cdot |T_{vs}| = \sum_{v \in T} (|T_{lc(v)}| + |T_{rc(v)}| + 1) \cdot |T_{vs}|$, and by Lemma 8, $\sum_{v \in T} (|T_{lc(v)}| + |T_{rc(v)}|) \cdot |T_{vs}| \leqslant 2n^2$. For $\sum_{v \in T} |T_{vs}|$, we can see that a node in T is counted at most $\log n$ times since for any T_v, only nodes in the subtree rooted at a child of v with fewer nodes are counted. Thus $\sum_{v \in T} |T_{vs}| \leqslant n \log n$ and the corollary follows. ◀

F How to compute feasible values and W_is

We first formally define what is a feasible $t_i^{\uparrow}$ value. The definition of a feasible $t_i^{\downarrow}$ value is similar. Let $t_i^{\uparrow} = (d_{i,1}^{\uparrow}, \ldots, d_{i,|X_i|}^{\uparrow})$. For any two components $d_{i,k}^{\uparrow}$ and $d_{i,l}^{\uparrow}$, the length of the shortest path from $v_{i,k}$ to s that goes through shortcut $(v_{i,l}^{\uparrow}, s)$ has to be at least $d_{i,k}^{\uparrow}$, otherwise $d_{i,k}^{\uparrow}$ would have a smaller value. Conversely, the length of the shortest path from $v_{i,l}$ to s that goes through shortcut $(v_{i,k}^{\uparrow}, s)$ has to be at least $d_{i,l}^{\uparrow}$. The definition follows.

▶ **Definition 17.** *Let $t_i^{\uparrow} = (d_{i,1}^{\uparrow}, \ldots, d_{i,|X_i|}^{\uparrow})$. $t_i^{\uparrow}$ is feasible if and only if for any $1 \leqslant k < l \leqslant |X_i|$, $d_G(v_{i,k}, v_{i,l}^{\uparrow}) + |v_{i,l}^{\uparrow}, s| \geqslant d_{i,k}^{\uparrow}$ and $d_G(v_{i,l}, v_{i,k}^{\uparrow}) + |v_{i,k}^{\uparrow}, s| \geqslant d_{i,l}^{\uparrow}$.*

To save space, we store feasible values of $t_i^{\uparrow}$ (and $t_i^{\downarrow}$) in a multilist. As a preprocessing step, we compute distances between any pair of vertices in G by using an all-pairs shortest paths algorithm. The distance between any pair of vertices in G can then be obtained in constant time. Let $\mathcal{L}_i^{\uparrow}$ denote the multilevel list for $t_i^{\uparrow}$. Levels of $\mathcal{L}_i^{\uparrow}$ are contain ordered values of $d_{i,1}^{\uparrow}, \ldots, d_{i,|X_i|}^{\uparrow}$, respectively. We store the sorted values of $\{d_G(v_{i,j}, u) + |us| | u \in V \backslash (X_{T_i} \backslash X_i)\}$ for each $v_{i,j} \in X_i$ in a list $l_j^{\uparrow}$, separately. We build $\mathcal{L}_i^{\uparrow}$ from $\{l_j^{\uparrow}\}$.

Besides $d_{i,j}^{\uparrow}$ value, each node of the first level of $\mathcal{L}_i^{\uparrow}$ stores a $(|X_i| - 1)$-level list. Each node of a second level of $\mathcal{L}_i^{\uparrow}$ stores a $(|X_i| - 2)$-level list, and so on. Generally, a node in a kth $(1 \leqslant k \leqslant |X_i|)$ level list stores a $(|X_i| - k)$-level list. A kth level list is built recursively as follows. Assume $d_{i,1}^{\uparrow}, \ldots, d_{i,k-1}^{\uparrow}$ are the distance values contained in nodes of previous levels. We build this kth level list by $l_k^{\uparrow}, \ldots, l_{|X_i|}^{\uparrow}$. For each $d_{i,k}^{\uparrow}$ value in $l_k^{\uparrow}$, we check whether

the condition in Definition 17 is satisfied between $d^{\uparrow}_{i,k}$ and each of $d^{\uparrow}_{i,1}, \ldots, d^{\uparrow}_{i,k-1}$. If so, we create a node with the current $d^{\uparrow}_{i,k}$ value, and build its $(|X_i| - k)$-level list recursively by $l^{\uparrow}_{k+1}, \ldots, l^{\uparrow}_{|X_i|}$. In this way, we can build $\mathcal{L}^{\uparrow}_i$ in $O(bn^b)$ time. An example $\mathcal{L}^{\uparrow}_i$ is shown in Figure 9(a). The $d^{\uparrow}_{i,2}$ value stored at a red node in a vertical (second level) list is realized by the same vertex as the $d^{\uparrow}_{i,1}$ value stored at its first level list node. Feasible values of $t^{\downarrow}_i$ are constructed in the same way. To access values of R_i fast, we can encode feasible $t^{\uparrow}_i$ and $t^{\downarrow}_i$ values as indices into a multidimensional array. We can then access R_i values in $O(b)$ time.

▶ **Lemma 18.** *For any node i of (X, T), the feasible values of $t^{\uparrow}_i$ ($t^{\downarrow}_i$) can be computed in $O(bn^b)$ time. R_i values can be accessed in $O(b)$ time.*

We now discuss how to compute a W_i. The feasible $t^{\downarrow}_i$ values for a W_i are just the feasible $t^{\downarrow}_i$ values for R_i. However, to deal with the components in $t^{\downarrow}_{i2}$, we build the multilist for $t^{\downarrow}_i$ by using components in $t^{\downarrow}_{i1}$ for the first $|t^{\downarrow}_{i1}|$ levels and using components in $t^{\downarrow}_{i2}$ for the remaining $|t^{\downarrow}_{i2}|$ levels. After building the multilist for $t^{\downarrow}_i$ in this order, we computes values of W_i in $|t^{\downarrow}_{i2}|$ rounds. Assume $t^{\downarrow}_{i2} = (d^{\downarrow}_{i,j_1}, \ldots, d^{\downarrow}_{i,j_s})$. The first round computes values such that $d^{\downarrow}_{i,j_1}, \ldots, d^{\downarrow}_{i,j_{s-1}}$ are fixed while $d^{\downarrow}_{i,j_s}$ is greater than or equal to the specified value. The second round computes values such that $d^{\downarrow}_{i,j_1}, \ldots, d^{\downarrow}_{i,j_{s-2}}$ are fixed while $d^{\downarrow}_{i,j_{s-1}}, d^{\downarrow}_{i,j_s}$ are greater than or equal to the specified values. And so on. By adding links between nodes in the multilist, each round build its result on the previous round. An example is shown in Figure 9(b). In the example, $t^{\downarrow}_{i2} = t^{\downarrow}_i$ and $|X_i| = 2$. The links added for the 2nd round are drawn in blue.

▶ **Lemma 19.** *For a given $t^{\uparrow}_i$, W_i values for all feasible $t^{\downarrow}_i$ and $\bar{k}$ can be computed in $O(kb^2n^b)$ time. All values of a W_i can be computed in $O(kb^2n^{2b})$ time.*

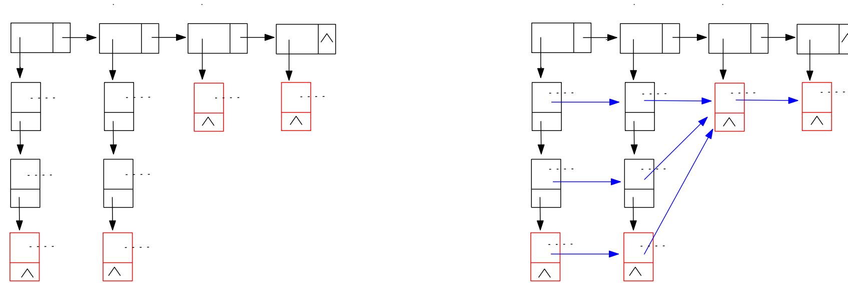

Figure 9 (a) $\mathcal{L}^{\downarrow}_i$ where $|X_i| = 2$. (b) Blue links between nodes in $\mathcal{L}^{\downarrow}_i$. The value stored in the pointed to node is greater than or equal to the value stored in the pointed from node.