Sum-of-Local-Effects Data Structures for Separable Graphs*

Xing Lyu[1], Travis Gagie[2][0000−0003−3689−327X], Meng He[2][0000−0003−0358−7102], Yakov Nekrich[3][0000−0003−3771−5088], and Norbert Zeh[2][0000−0002−−0562−1629]

[1] Halifax West High School, Halifax, Canada lyuxing1006@gmail.com
[2] Dalhousie University, Halifax, Canada {firstname.lastname}@dal.ca
[3] Michigan Technological University yakov@mtu.edu

Abstract. It is not difficult to think of applications that can be modelled as graph problems in which placing some facility or commodity at a vertex has some positive or negative effect on the values of all the vertices out to some distance, and we want to be able to calculate quickly the cumulative effect on any vertex's value at any time or the list of the most beneficial or most detrimential effects on a vertex. In this paper we show how, given an edge-weighted graph with constant-size separators, we can support the following operations in time polylogarithmic in the number of vertices and the number of facilities placed on the vertices, where distances between vertices are measured with respect to edge weights:

Add(v, f, w, d) places a facility of weight w and with effect radius d onto vertex v.

Remove(v, f) removes a facility f previously placed on v using Add from v.

Sum(v) or Sum(v, d) returns the total weight of all facilities affecting v or, with a distance parameter d, the total weight of all facilities whose effect region intersects the "circle" with radius d around v.

Top(v, k) or Top(v, k, d) returns the k facilities of greatest weight that affect v or, with a distance parameter d, whose effect region intersects the "circle" with radius d around v.

The weights of the facilities and the operation that Sum uses to "sum" them must form a semigroup. For Top queries, the weights must be drawn from a total order.

Keywords: Graph data structures · Treewidth · Branchwidth · Graph decompositions · Tree decompositions · Sum of local effects

1 Introduction

Even people who have never heard of Baron Samuel of Wych Cross may have heard a saying often attributed to him, that there are three things that matter in real estate: location, location, location. This means that the value of a property may increase or decrease depending on whether it is close to a bus stop, a good

* This work is supported by NSERC.

school, a supermarket or a landfill, for example. Of course, "close" may not mean the same thing for a bus stop as it does for a landfill, and the positive effect of the former may not offset the negative effect of the latter. In fact, "close" may not refer to Euclidean distance, since walking to a bus stop five minutes down the street is preferable to walking to one five minutes away through a landfill. To model applications in which there are such additive local effects with a non-Euclidean definition of locality, we propose in this paper a data structure for a graph G that supports the following operations:

ADD(v, f, w, d) places a facility of weight w onto vertex v. The *effect region* of f is a circle with radius d around v.

REMOVE(v, f) removes a facility f previously placed on v using ADD from v.

SUM(v) or SUM(v, d) returns the total weight of all facilities affecting v or, with a distance parameter d, the total weight of all facilities whose effect region intersects the "circle" with radius d around v.

TOP(v, k) or TOP(v, k, d) returns the k facilities of greatest weight that affect v or, with a distance parameter d, whose effect region intersects the "circle" with radius d around v.

We assume that every edge $e \in G$ has a non-negative length $\ell(e)$ and that distances between vertices are measured as the minimum total length of all edges on any path between these two vertices. A circle with radius d around some vertex v includes all vertices and (parts of) edges at distance d from v. More precisely, if f is a facility with effect radius d' placed on some vertex u, then we consider f's effect region to intersect a circle with radius d around some other vertex v if and only if $\text{dist}(u, v) \leq d + d'$.

The weights of the facilities and the operation that SUM uses to "sum" them must form a semigroup. For TOP queries, the weights must be drawn from a total order. Note that SUM(v) and TOP(v, k) can be viewed as "range stabbing queries on graphs", whereas SUM(v, d) and TOP(v, k, d) with $d > 0$ are "range intersection queries on graphs," where the ranges are the effect regions of the facilities and a query is either an individual vertex or a region of some radius d around some vertex.

We call such a data structure a sum-of-local-effects (SOLE) data structure. In Section 2, we show that when G is a tree on n vertices, then there is a SOLE data structure for it supporting ADD, REMOVE, and SUM operations in $O(\lg n \lg m)$ time, and TOP queries in $O(k \lg n \lg m)$ time, where m is the total number of facilities currently placed on the vertices of G. In Section 3, we generalize this result to t-separable graphs, for any constant t, which includes series-parallel graphs ($t \leq 2$), graphs of constant treewidth w ($t \leq w + 1$), and graphs of constant branchwidth b ($t \leq b$). We show that when G is t-separable, there exists a SOLE data structure for it supporting ADD, REMOVE, and SUM operations in $O(\lg n \lg^t m)$ time, and TOP queries in $O(k \lg n \lg^t m)$ time. The costs of ADD and REMOVE operations are amortized in this case.

Our results can be extended to *directed* graphs G and our data structure can be made to support vertex and edge deletions in G. We will investigate this generalization in the full version of this paper.

2 A SOLE Data Structure for Trees

In this section, we prove that

Theorem 1. *If G is a tree on n vertices, then there is a SOLE data structure for it supporting* Add, Remove, *and* Sum *operations in $O(\lg n \lg m)$ time, and* Top(v, k, d) *operations in $O(k \lg n \lg m)$ time, where m is the number of facilities currently on the vertices of G. The size of this data structure is $O(n + m \lg n)$.*

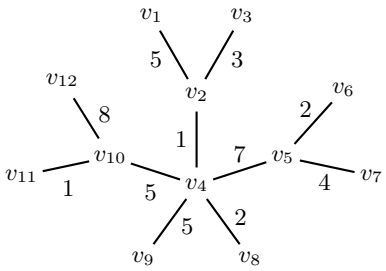
To obtain a SOLE data structure for arbitrary trees, we can transform any tree G into a tree G' whose nodes have degree at most 3 by replacing every high-degree vertex u in G with a degree-3 subtree G'_u whose edges all have length 0 (see Figs. 1b,c). We choose an arbitrary vertex in G'_u as the representative of u in G'. This ensures that the distances between vertices in G and between their representatives in G' are the same. Thus, we can support operations on G by building a SOLE data structure on G' instead. Therefore, for the rest of this section, we assume that all vertices of G have degree at most 3.

2.1 Designing SOLE Data Structures for Trees

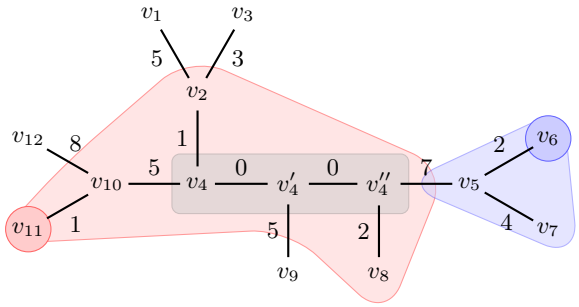
We choose an arbitrary vertex ρ of G and label every vertex v in G with its distance $\text{dist}(\rho, v)$ from ρ. A *centroid edge* of G is an edge (u, v) whose removal splits G into two subtrees G_u and G_v with at most $2n/3$ vertices each. Such an edge exists because all vertices of G have degree at most 3. A *centroid decomposition* of G is a binary tree T defined inductively as follows (see Fig. 1c): If G has a single vertex v, then T has v as its only node. Otherwise, let (u, v) be an arbitrary centroid edge of G. Then the root of T is (u, v), and the two children of (u, v) are the roots of centroid decompositions of G_u and G_v. For each edge e of T, let T_e be the subtree of T below e, and let V_e be the set of vertices of G corresponding to the leaves of T_e. The height of T is at most $\log_{3/2} n = O(\lg n)$.

Our SOLE data structure for G consists of a centroid decomposition T of G where each edge e of T has an associated data structure W_e storing facilities in $V_{e'}$ that may affect the vertices in V_e, where e' is the other edge in T descending from the same node as e. Each leaf v of T (corresponding to the vertex v of G) also has an associated data structure W_v storing the facilities placed on v itself. Each facility f with weight w in W_e or W_v has an associated *radius* r and is stored as the triple (r, f, w) in W_e.

We represent each data structure W_x, where x can be an edge or a leaf of T, as two search trees R_x and F_x. R_x is a priority search tree [4] on the triples (r, f, w) in W_x, using the radii r as x-coordinates and the weights w as y-coordinates. Each node v of R_x is augmented with the total weight of all triples in the subtree below v. F_x is a standard search tree over the triples (r, f, w) in W_x, using the identifiers f of facilities as keys. The two copies of (r, f, w) in R_x and F_x are linked using cross pointers. Thus, W_x supports the following operations in $O(\lg m)$ time: insertion of a new triple (r, f, w), deletion of a triple associated with facility f, and reporting of the total weight of all triples (r, f, w)

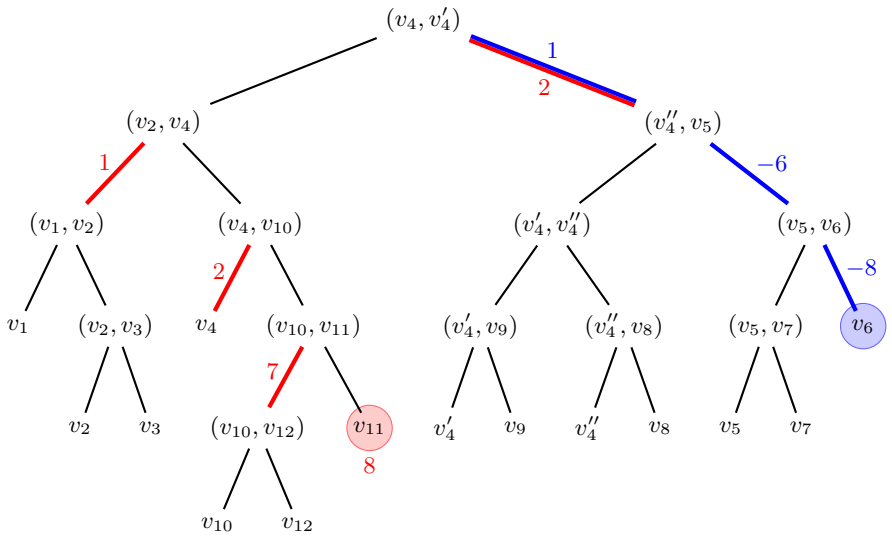


(a)



(b)

Fig. 1: A tree (a), its degree-3 version (b), and the centroid decomposition (c, next page) of the tree in (b). The shaded subtree in (b) is the degree-3 tree replacing the high-degree vertex v_4 in (a). A facility f with effect radius 8 placed on v_{11} has the pink effect region in (b). This region overlaps the blue query region with radius 8 around v_6. In the centroid decomposition (c), f is stored in the edge data structures of the fat red edges and the node data structure of v_{11}, with the radii shown in red. A query with radius 8 around v_6 queries the node data structure of v_6 an the edge data structures of the fat blue edges, with the query radii shown in blue. In particular, f is reported as part of the query on the data structure W_e associated with the highlighted child edge of (v_4, v_4').



(c)

Fig. 1 continued

with $q \leq r$, for some query radius q. It also supports, in $O(k \lg m)$ time, reporting the k triples with maximum weight among all triples (r, f, w) with $q \leq r$.

To bound the size of the data structure, note that T has size $O(n)$, each data structure W_x, with x a leaf or edge of T, has size linear in the number of triples it stores, and each facility placed on some vertex v is stored in W_v and in the data structure W_e associated with each edge e on the path $P_v = \langle x_1, \ldots, x_h, v \rangle$ from the root of T to v (which is a leaf of T). Since this path has length $O(\lg n)$, each facility is stored $O(\lg n)$ times. Thus, the SOLE data structure for trees has size $O(n + m \lg n)$.

2.2 Supporting Queries over Trees

We now use our data structures to support operations.

An $\textsc{Add}(v, f, w, d)$ operation traverses the path P_v. We insert the triple (d, f, w) into W_v. Each node x_i in P_v represents an edge (x, y) of G and has two child edges e_x and e_y such that $x \in V_{e_x}$ and $y \in V_{e_y}$. Assume w.l.o.g. that $v \in V_{e_x}$, and let $d' = |\text{dist}(\rho, y) - \text{dist}(\rho, v)|$. We insert the triple $(d - d', f, w)$ into the data structure W_{e_y} associated with e_y. This is illustrated in Fig. 1c. This takes $O(\lg m)$ time per vertex in P_v, $O(\lg n \lg m)$ time in total.

A $\textsc{Remove}(v, f)$ operation traverses the path P_v and deletes the triple associated with f from the data structure W_e of every child edge e of every node x_i in P_v, and from W_v. By a similar analysis as for $\textsc{Add}(v, f, w, d)$ operations, this takes $O(\lg n \lg m)$ time.

A $\textsc{Sum}(v, d)$ query traverses the path P_v. For each edge e on this path with top endpoint x_i, x_i represents an edge $(x, y) \in G$ such that w.l.o.g. $e = e_y$. We query W_e to report the total weight of all triples (r, f, w) in W_e with $r \geq |\text{dist}(\rho, y) - \text{dist}(\rho, v)| - d$. We also query W_v to report the total weight of all facilities placed on v itself. This is illustrated in Fig. 1c. We sum these weights retrieved from W_v and from all the edge data structures W_e along P_v and report the resulting total. Since we answer a $\textsc{Sum}(v, d)$ query by querying $O(\lg n)$ data structures W_e and W_v, the cost is $O(\lg n \lg m)$. A $\textsc{Sum}(v)$ query is the same as a $\textsc{Sum}(v, 0)$ query.

A $\textsc{Top}(v, k, d)$ query traverses P_v. For each edge e on this path with top endpoint x_i, x_i represents an edge $(x, y) \in G$ such that w.l.o.g. $e = e_y$. We query W_e to retrieve the k triples with maximum weight among all triples (r, f, w) in W_e such that $r \geq |\text{dist}(\rho, y) - \text{dist}(\rho, v)| - d$. This takes $O(k \lg n \lg m)$ time for all edges on P_v. We also retrieve the k facilities with maximum weight from W_v, which takes $O(k \lg m)$ time. The k maximum-weight facilities affecting vertices at distance at most d from v are among the $O(k \lg n)$ facilities retrieved by these queries and can be found in $O(k \lg n)$ time using linear-time selection [2]. Thus, a $\textsc{Top}(v, k, d)$ query takes $O(k \lg n \lg m)$ time. A $\textsc{Top}(v, k)$ query is the same as a $\textsc{Top}(v, k, 0)$ query.

To prove the correctness of $\textsc{Sum}(v, d)$ and $\textsc{Top}(v, k, d)$ queries, note that both queries query the same data structures W_e, with the same query regions. A Sum query reports the total weight of all facilities in these query regions. A Top query reports the k maximum weight queries in these regions. Both queries are correct if we can argue that if either query were to *report* all facilities in these query regions instead of summing their weights or picking the k facilities with maximum weight, then any facility placed on some vertex u is reported if and only if its effect radius d' satisfies $d + d' \geq \text{dist}(u, v)$, and any such facility is reported exactly once.

So consider a facility f with effect radius d' placed on some vertex $u \in G$, and let v be another vertex $v \in G$. If $v = u$, then f must be reported because it affects v no matter the query radius d. The facility f does not belong to any data structure W_e on the path $P_u = P_v$. Thus, if f is to be reported, it must be reported by the query W_v. Since placing f on u adds f to $W_u = W_v$, a $\textsc{Sum}(v, d)$ query reports the total weight of all facilities in W_v, and a $\textsc{Top}(v, k, d)$ query reports the k facilities with maximum weight in W_v, the corresponding reporting query would report all facilities in W_v, including f.

If $v \neq u$, then let x_i be the highest vertex on the path from u to v in T. This vertex represents an edge (x, y) such that w.l.o.g. $u \in V_{e_x}$ and $v \in V_{e_y}$. In this case, f is not stored in W_v, and e_y is the only edge on the path P_v that is a pendant edge of P_u. Thus, W_{e_y} is the only data structure considered by a $\textsc{Sum}(v, d)$ or $\textsc{Top}(v, k, d)$ query that stores f. In W_{e_y}, f is stored with radius $r = d' - |\text{dist}(\rho, y) - \text{dist}(\rho, u)| = d' - \text{dist}(u, y)$. The path from u to v in G passes through y, so $\text{dist}(u, v) = \text{dist}(u, y) + \text{dist}(y, v)$. Therefore, $\text{dist}(u, v) \leq d + d'$ if and only if $q = |\text{dist}(\rho, v) - \text{dist}(\rho, y)| - d = \text{dist}(v, y) - d \leq d' - \text{dist}(u, y) = r$.

The reporting version of a $\textsc{Sum}(v, d)$ or $\textsc{Top}(v, k, d)$ query reports all triples (r, f, w) in W_{e_y} with $r \geq q$. Thus, f is reported if and only if $d + d' \geq \text{dist}(u, v)$.

This finishes the proof of Theorem 1.

3 A SOLE Data Structures for Separable Graphs

We call a graph G *t-separable*, for some constant t, if it has a *t-separator decomposition* C of the following structure (see Figs. 2a,b):

- C is an unrooted tree with $O(n)$ nodes, all of which have degree at most 3.
- Each edge e of C has an associated subset $S_e \subseteq V$ of vertices of G of size $|S_e| \leq t$. We call S_e the *(edge) bag* associated with e.
- Every vertex of G belongs to at least one bag of C.
- Let C_1 and C_2 be the subtrees of C obtained by removing any edge e from C, and let V_i, $i \in \{1, 2\}$, be the union of the bags of all edges in C_i. Then any path from a vertex in V_1 to a vertex in V_2 includes at least one vertex in S_e. In other words, S_e separates the vertices in V_1 from the vertices in V_2.

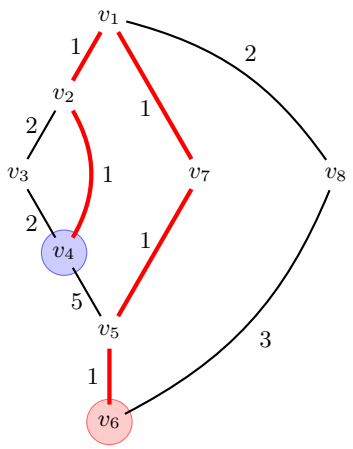
This definition of a t-separator decomposition is similar to both a tree decomposition [5] and a branch decomposition [5], but the properties of a t-separator decomposition are weaker than those of both a tree-decomposition and a branch decomposition. In particular, a branch decomposition of width b and degree 3 is easily seen to be a b-separator decomposition, and a nice tree decomposition C [3] of width w gives rise to a $(w+1)$-separator decomposition by defining the (edge) bag associated with each edge $(v, w) \in C$ to be the union of the (node) bags associated with v and w. However, a t-separator decomposition does not require a bijection between the edges of G and the leaves of C, as required by a branch decomposition. A tree decomposition requires that for every edge (v, w) of G, there exists a (node) bag that contains both v and w, a condition not imposed by a t-separator decomposition. Thus, every graph of branchwidth b has a t-separator decomposition with $t \leq b$, every graph of treewidth w has a t-separator decomposition with $t \leq w + 1$, but there may exist graphs for which these inequalities are strict.

In this section, we prove that

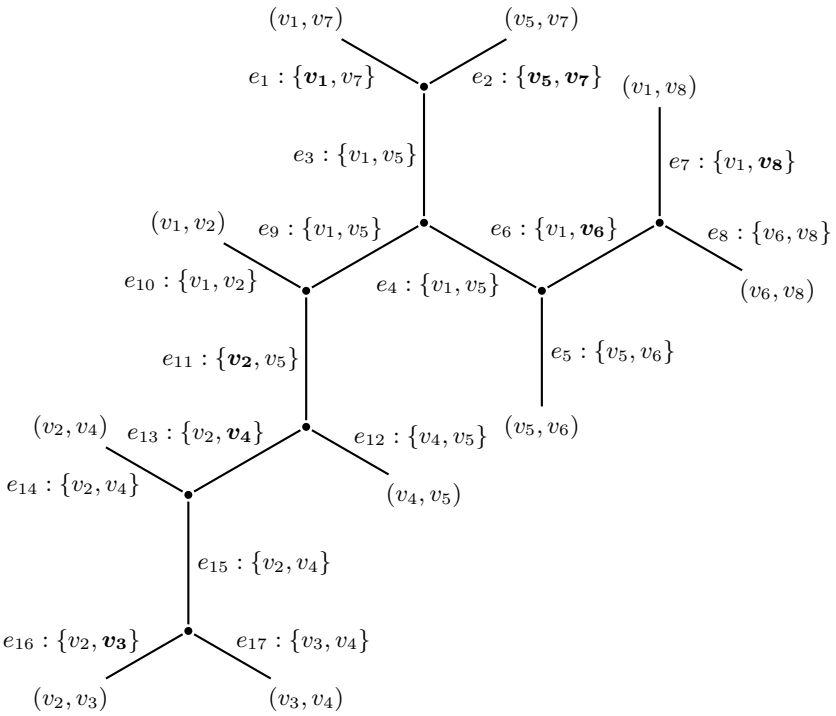
Theorem 2. *If G is a t-separable graph on n vertices, for some constant t, then there is a SOLE data structure for it supporting* Add, Remove, *and* Sum *operations in $O(\lg n \lg^t m)$ time, and $\textsc{Top}(v, k, d)$ operations in $O(k \lg n \lg^t m)$ time. The size of this data structure is $O(tn \lg n + m \lg^{t-1} m \lg n)$. The costs of* Add *and* Remove *operations are amortized.*

3.1 Designing SOLE Data Structures for Separable Graphs

To design our SOLE structure, let C be a t-separator decomposition for G. Since C is an unrooted tree whose nodes have degree at most 3, we can once again construct its centroid decomposition T (see Fig. 2c). Each leaf of T corresponds

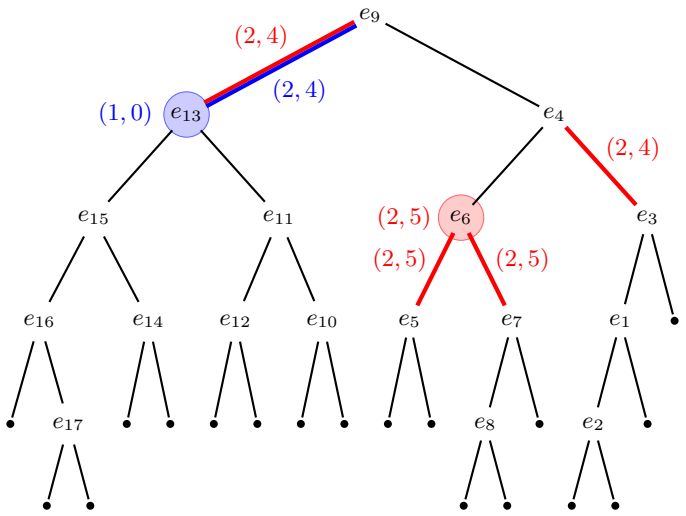


(a)



(b)

Fig. 2: Caption on next page.



(c)

Fig. 2 continued: A series-parallel graph G (a), a branch decomposition C of G of width 2 (b), which is also a 2-separator decomposition of G, and the centroid decomposition T of C (c). Each edge in (a) is labelled with its length. Each edge in (b) is labelled with its name and its corresponding bag S_e. A facility with effect radius 5 placed on vertex v_6 affects vertex v_4, since the path shown in red in (a) has length 5. If we assume that $e_{v_6} = e_6$ and $e_{v_4} = e_{13}$, as indicated by the bold vertices in (b), then f is stored in the node data structure of e_6 and in the edge data structures of the red edges in (c), with the pairs of radii shown in red. A $\textsc{Sum}(v_4)$ or $\textsc{Top}(v_4, k)$ query queries the vertex data structure of e_{13} and the edge data structure of the blue edge in (c), with the pairs of radii shown in blue. The facility f is added to the query result because $2 \leq 2$ and $4 \leq 4$. One of these two conditions would have sufficed.

to a node of C, and each internal node of T corresponds to an edge e of C. Since C has size $O(n)$, the height of T is $O(\lg n)$. Since a vertex v of G may be contained in more than one bag of C, we choose an arbitrary bag S_e of C that contains v and refer to e as e_v, as indicated by the bold vertex labels in Fig. 2b.

We obtain a SOLE data structure for G by augmenting each internal node e of T with two data structures W_e and D_e and augmenting each edge a of T with a data structure W_a. We refer to W_e as a *node data structure* and to W_a as an *edge data structure*. Each data structure W_x, with x a node or an edge of T, stores a number of facilities f as tuples $(r_1, \ldots, r_{t'}, f, w)$. If $x = e$ is a node of T or $x = a$ is an edge of T with top endpoint e, then $t' = |S_e| \leq t$. Again, W_x consists of two trees R_x and F_x over the set of tuples stored in W_x. F_x stores these tuples as a binary search tree with the facility f as the key for each tuple $(r_1, \ldots, r_{t'}, f, w)$ in W_x. R_x is a "t'-dimensional range sum priority

search tree" over the points defined by the coordinates $(r_1, \ldots, r_{t'})$. This is a t'-dimensional range tree [1, 6] augmented to support t'-dimensional range sum queries in $O\big(\lg^{t'} m\big)$ time and t'-dimensional range top-k queries in $O\big(k \lg^{t'} m\big)$ time.

The structure D_e associated with each internal node e of T stores the distance from every vertex v such that e_v is a descendant of e in T to all vertices in S_e.

A t'-dimensional range sum priority search tree supports insertions and deletions in $O\big(\lg^{t'} m\big)$ amortized time. Thus, each data structure W_x supports insertion of a new tuple $(r_1, \ldots, r_{t'}, f, w)$ and the deletion of the tuple associated with a facility f in $O\big(\lg^{t'} m\big) = O(\lg^t m)$ amortized time.

We now bound the size of the SOLE data structure. Once again, the tree T has size $O(n)$ and height $O(\lg n)$. For every vertex $v \in G$, the distance data structure D_e of every ancestor node e of e_v in T stores the distances from v to all $t' \leq t$ vertices in S_e. Thus, each vertex v contributes at most t to the size of each of $O(\lg n)$ distance data structures. The total size of the distance data structures is thus $O(tn \lg n)$. Each facility f placed on some vertex u is stored in the node data structures of the $O(\lg n)$ nodes along the path P_u and in one or two edge data structures of child edges of these nodes. Thus, every facility is stored in $O(\lg n)$ data structures W_x. Each such data structure is a t'-dimensional range sum priority search tree, where $t' \leq t$. Thus, if it stores s facilities, it has size $O(s \lg^{t-1} s)$. The total size of all node and edge data structures is thus $O(m \lg^{t-1} m \lg n)$.

3.2 Sum and Top-k Over W_x with an Uncommon Query Range

To support Sum and Top, we need to support range sum queries and range top-k queries on W_x, but with an uncommon query range. A query point $q = (q_1, \ldots, q_{t'})$ defines a query range $\mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$, i.e., the complement of t'-sided range query. We need to support range sum queries and range top-k queries for any such complement of a t'-sided range query.

An easy solution is to decompose it into t' "normal" range queries, with query ranges $[q_1, \infty) \times (-\infty, \infty) \times \cdots \times (-\infty, \infty)$, $(-\infty, q_1) \times [q_2, \infty) \times (-\infty, \infty) \times \cdots \times (-\infty, \infty)$, $(-\infty, q_1) \times (-\infty, q_2) \times [q_3, \infty) \times (-\infty, \infty) \times \cdots \times (-\infty, \infty)$, $\cdots$, $(-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'-1}) \times [q_{t'}, \infty)$. This allows us to support range sum and range top-k queries in $O\big(t' \lg^{t'} m\big)$ and $O\big(t' k \lg^{t'} m\big)$ time, respectively, which is $O\big(\lg^{t'} m\big)$ and $O\big(k \lg^{t'} m\big)$ time because $t' \leq t$ and t is a constant.

A better way to support range queries with query ranges of the form $\mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$ without the factor t' overhead is to implement them directly on the t'-dimensional range sum priority search tree. To answer a range sum query with such a query range, we answer a 1-dimensional range sum query with query range $[q_1, \infty)$ on the level-1 tree of R_x. This query traverses the path corresponding to q_1 in the level-1 tree of R_x. For the root of each subtree to the left of this path, we answer a $(t'-1)$-dimensional range sum query with query range $\mathbb{R}^{t'-1} \setminus ((-\infty, q_2) \times (-\infty, q_3) \times \cdots \times (-\infty, q_{t'}))$ on the $(t'-1)$-dimensional range sum priority search tree associated with this root.

The final result is the sum of the totals produced by these queries, including the 1-dimensional range sum query on the level-1 tree of R_x. Thus, a range sum query with the complement of a t'-sided range query as the query range has the same cost as a "normal" orthogonal range sum query, $O\big(\lg^{t'} m\big)$.

Similarly, to support a t'-dimensional range top-k query with query range $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$, we answer a 1-dimensional range top-k query on the level-1 tree of R_x, with query range $[q_1, \infty)$. This query traverses the path corresponding to q_1 in the level-1 tree of R_x. For the root of each subtree to the left of this path, we answer a $(t'-1)$-dimensional range top-k query with query range $\mathbb{R}^{t'-1} \setminus ((-\infty, q_2) \times (-\infty, q_3) \times \cdots \times (-\infty, q_{t'}))$ on the $(t'-1)$-dimensional range sum priority search tree associated with this root. The top k tuples in the query range Q are easily seen to be among the $O(k \log n)$ elements reported by these $(t'-1)$-dimensional range top-k queries and by the 1-dimensional range top-k query on the level-1 tree. The top k tuples can now be found in $O(k \lg n)$ time using linear-time selection [2]. Thus, a range top-k query with the complement of a t'-sided range query as the query range takes $O\big(k \lg^{t'} m\big)$ time, just as a "normal" orthogonal range top-k query does.

3.3 Supporting Queries over Separable Graphs

We are ready to discuss how to support Sum and Top queries for t-separable graphs; the support for Add and Remove will be described in the full version of this paper.

A $\textsc{Sum}(v, d)$ query traverses the path $P_v = \langle e_1, \ldots, e_h = e_v \rangle$. For each node e_i in P_v, let $S_{e_i} = \{v_1, \ldots, v_{t'}\}$, and let $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$, where $q_j = \text{dist}(v, v_i) - d$, for all $1 \le j \le t'$. If $1 \le i < h$, then we answer a range sum query with query range Q on the edge data structure W_a, where $a = (e_i, e_{i+1})$ is the child edge of e_i that belongs to P_v. If $i = h$, then we answer a range sum query with query range Q on W_{e_i}. This is illustrated in Fig. 2c. The result of the $\textsc{Sum}(v, d)$ query is the sum of the results reported by all these range sum queries. Since a range sum query on each data structure W_x can be answered in $O\big(\lg^{t'} m\big) = O(\lg^t m)$ time, the cost of a $\textsc{Sum}(v, d)$ query is thus $O(\lg n \lg^t m)$. A $\textsc{Sum}(v)$ query is the same as a $\textsc{Sum}(v, 0)$ query.

A $\textsc{Top}(v, k, d)$ query traverses the path $P_v = \langle e_1, \ldots, e_h = e_v \rangle$. For each node e_i in P_v, let $S_{e_i} = \{v_1, \ldots, v_{t'}\}$, and let $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$, where $q_j = \text{dist}(v, v_i) - d$, for all $1 \le j \le t'$. If $1 \le i < h$, then we ask a range top-k query with query range Q on the edge data structure W_a, where $a = (e_i, e_{i+1})$ is the child edge of e_i that belongs to P_v. If $i = h$, then we answer a range top-k query with query range Q on W_{e_i}. The result of the $\textsc{Top}(v, k, d)$ query is the list of the k maximum-weight facilities among the $O(k \lg n)$ facilities reported by all these range top-k queries. These k facilities can be found in $O(k \lg n)$ time using linear-time selection [2]. Each query on a data structure W_x takes $O(k \lg^t m)$ time. Thus, the total cost of a $\textsc{Top}(v, k, d)$ query is $O(k \lg n \lg^t m)$. A $\textsc{Top}(v, k)$ query is the same as a $\textsc{Top}(v, k, 0)$ query.

To establish the correctness of $\textsc{Sum}(v, d)$ and $\textsc{Top}(v, k, d)$ queries, observe that, similar to Section 2, both queries query the same node and edge data

structures, with the same query ranges. Thus, it suffices to prove that if either query reported all facilities in these query ranges, it would report any facility with effect radius d' placed on some vertex u if and only if $d + d' \geq \text{dist}(u, v)$, and each such facility is reported exactly once.

So let f be a facility with effect radius d' placed on some vertex $u \in G$, and let v by any other vertex $v \in G$. Let e be the lowest common ancestor (LCA) of e_u and e_v in T, and let $S_e = \{v_1, \ldots, v_{t'}\}$ We distinguish two cases:

If e_v is a proper descendant of e, then f is not stored in W_{e_v} and the only edge data structure in P_v that stores f is the data structure W_a corresponding to the child edge a of e on the path from e to e_v. Thus, f is reported at most once by the reporting version of a $\textsc{Sum}(v, d)$ or $\textsc{Top}(v, k, d)$ query. This query queries W_a with query region $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times \cdots (-\infty, q_{t'}))$, where $q_j = \text{dist}(v_j) - d$ for all $1 \leq j \leq t'$. Since e is the LCA of e_u and e_v in T, any path from u to v in G must include at least one vertex in S_e. Assume w.l.o.g. that v_1 is one such vertex. Then $\text{dist}(u, v) = \text{dist}(v, v_1) + \text{dist}(u, v_1)$ and $\text{dist}(u, v) \leq \text{dist}(v, v_j) + \text{dist}(u, v_j)$ for all $1 < j \leq t'$. The facility f is stored in W_a as the tuple $(r_1, \ldots, r_{t'}, f, w)$ with $r_j = d' - \text{dist}(u, v_j)$ for all $1 \leq j \leq t'$. Thus, $(r_1, \ldots, r_{t'}) \in Q$ if and only if there exists an index $1 \leq j \leq t'$ such that $d' - \text{dist}(u, v_j) \geq \text{dist}(v, v_j) - d$, that is, $d + d' \geq \text{dist}(u, v_j) + \text{dist}(v, v_j)$. Since $\text{dist}(u, v_1) + \text{dist}(v, v_1) = \text{dist}(u, v)$ and $\text{dist}(u, v_j) + \text{dist}(v, v_j) \geq \text{dist}(u, v)$ for all $1 \leq j \leq t'$, this is true if and only if $d + d' \geq \text{dist}(u, v)$. Thus, f is reported if and only if $d + d' \geq \text{dist}(u, v)$.

If e_v is not a proper descendant of e, then $e_v = e$ and $P_v \subseteq P_u$. Thus, f is not stored in any edge data structure along P_v, but it is stored in $W_e = W_{e_v}$, as the tuple $(r_1, \ldots, r_{t'}, f, w)$ with $r_j = d' - \text{dist}(u, v_j)$ for all $1 \leq j \leq t'$. Thus, f is reported at most once by the reporting version of a $\textsc{Sum}(v, d)$ or $\textsc{Top}(v, k, d)$ query. This query queries W_{e_v} with query region $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times \cdots (-\infty, q_{t'}))$, where $q_j = \text{dist}(v_j) - d$ for all $1 \leq j \leq t'$. Since $v \in S_{e_v}$, we can assume w.l.o.g. that $v = v_1$. Then $\text{dist}(u, v) = \text{dist}(v, v_1) + \text{dist}(u, v_1)$ and $\text{dist}(u, v) \leq \text{dist}(v, v_j) + \text{dist}(u, v_j)$ for all $1 < j \leq t'$. The same analysis as in the previous case now shows that f is reported by the query on W_{e_v} if and only if $d + d' \geq \text{dist}(u, v)$. This finishes the proof of Theorem 2.

References

1. Bentley, J.L.: Decomposable searching problems. Information Processing Letters **8**(5), 244–251 (1979)
2. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. Journal of Computer and System Sciences **7**(4), 448–461 (1973)
3. Bodlaender, H.L., Kloks, T.: Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs. Journal of Algorithms **21**(2), 358–402 (1996)
4. McCreight, E.M.: Priority Search Trees. SIAM Journal on Computing **14**(2), 257–276 (1985)
5. Robertson, N., Seymour, P.D.: Graph minors. X. Obstructions to tree-decomposition. Journal of Combinatorial Theory, Series B **52**(2), 153–190 (1991)
6. Willard, D.E., Lueker, G.S.: Adding range restriction capability to dynamic data structures. Journal of the ACM **32**(3), 597–617 (1985)