Dynamic Tree Cross Products

Marcus Raitner

University of Passau, D-94032 Passau, Germany,
`Marcus.Raitner@Uni-Passau.De`

Abstract. *Range searching over tree cross products* – a variant of classic range searching – recently has been introduced by Buchsbaum et al. (*Proc. 8th ESA*, vol. 1879 of *LNCS*, pp. 120–131, 2000). A tree cross product consist of hyperedges connecting the nodes of trees $T_1, \dots, T_d$. In this context, range searching means to determine all hyperedges connecting a given set of tree nodes. Buchsbaum et al. describe a data structure which supports, besides queries, adding and removing of edges; the tree nodes remain fixed. In this paper we present a new data structure, which additionally provides insertion and deletion of leaves of $T_1, \dots, T_d$; it combines the former approach with a novel technique of using search trees superimposed over ordered list maintenance structures. The extra cost for this dynamization is roughly a factor of $\mathcal{O}(^{\log n}/_{\log \log n})$. The trees being dynamic is especially important for *maintaining hierarchical graph views*, a problem that can be modeled as tree cross product. Such views evolve from a large base graph by the contraction of subgraphs defined recursively by an associated hierarchy. The graph view maintenance problem is to provide methods for refining and coarsening a view. In previous solutions only the edges of the underlying graph were dynamic; with the application of our data structure, the node set becomes dynamic as well.

1 Motivation

Many graphs, such as network traffic graphs, the web graph, or biochemical pathways [1], are too large to display or edit them effectively. A well-established technique to solve this problem is to partition the graph recursively into a hierarchy of subgraphs. The complete road map of Europe, for instance, is a rather large graph; a hierarchy on it can be defined, for instance, by grouping places and roads within the same city, then the cities within the same state, and so on. Not every city or state is always needed in full detail; the dispensable subgraphs, therefore, are contracted into a single *meta node*, representing the city or state as a whole. Edges from within the contracted subgraph to nodes outside are retained as edges from the meta node to the outside place. This leads to an abstract representation of the graph, a *graph view*, which is very convenient, because both an overview of the whole graph and the necessary details are displayed simultaneously.

In an interactive scenario, this facilitates exploring and editing a large graph: the user can choose which subgraphs to contract into meta nodes and which to expand, i. e., to replace with its subordinate subgraphs. Depending on the admissible modifications of the hierarchy and the graph, Buchsbaum and Westbrook [2] differentiate three variants of this *graph view maintenance problem*: in the *static* case the graph and the hierarchy are both fixed; in the *dynamic graph* variant graph edges can be inserted or deleted; in the *dynamic graph and tree* variant the graph additionally is subject to node insertions and deletions, and the hierarchy may change through splitting and merging of clusters.

As shown in [3] and briefly recapitulated in Sect. 3, maintaining hierarchical graph views can be formulated as a special case of *range searching over tree cross products*. A tree cross product consists of disjoint trees $T_1, \ldots, T_d$ and a set of d-dimensional hyperedges $\boldsymbol{u} = (u_1, \ldots, u_d)$ with $u_i \in T_i$ for $i = 1, \ldots, d$. In this context, range searching means to decide efficiently whether there is a hyperedge between the subtrees of given tree nodes and to report all those hyperedges. In [3] the set of hyperedges is dynamic, but the trees are static; we generalize this approach and allow insertion and deletion of leaves of $T_1, \ldots, T_n$ as well. After a formal definition of the problem, the data structure for the two-dimensional tree cross product is described in Sect. 2.1, which then serves as basis for the recursive description of the higher dimensional case in Sect. 2.2.

In Sect. 3 we introduce the new *dynamic leaves* variant of the graph view maintenance problem for *compound graphs* [4], a more general model of hierarchically structured graphs than the *clustered graphs* [5] used in previous approaches [2,3]. Formulated as a tree cross product, this problem can be solved efficiently with our data structure. The new dynamic leaves variant extends the dynamic graph variant by insertion and deletion of graph nodes, i. e., leaves of the hierarchy. In contrast to the dynamic graph and tree variant, it lacks splitting and merging of clusters. Thus, it is adequate for dynamic graphs with a fixed hierarchical structure, such as network traffic graphs: computers are identified by their IP addresses, edges represent traffic, and the hierarchy is given by the structure of the IP addresses. The hierarchy is fixed, because the structure of an IP address does not change. Further examples are road maps or biochemical reaction networks [1], which consist of all reactions in some organism; the hierarchy is often defined by grouping reactions that belong to the same biochemical pathway, e. g., the citric acid cycle. Unable to handle graphs with a dynamic node set, previous solutions [2,3] are not appropriate models for these applications.

2 Range Searching over Tree Cross Products

Let $T = (V(T), E(T))$ be a rooted tree with node set $V(T)$ and edge set $E(T)$. For a node $v \in V(T)$, let $\mathrm{children}(v)$ denote the set of all *children* of v and $\mathrm{parent}(v)$ the *parent* of v. The *descendants* of v, $\mathrm{desc}(v)$, are all nodes in the subtree rooted at v. Conversely, v is an *ancestor* of each $u \in \mathrm{desc}(v)$. Given d disjoint, rooted trees $T_1, \ldots, T_d$, consider a d-partite hypergraph G such that $V(G) = \bigcup_{i=1}^{d} V(T_i)$ and $E(G) \subseteq \prod_{i=1}^{d} V(T_i) = V(T_1) \times \cdots \times V(T_d)$. We define $n = |V(G)|$, $m = |E(G)|$, and $D = \max_{i=1}^{d} \mathrm{depth}(T_i)$. For a tuple $\boldsymbol{u} = (u_1, \ldots, u_d) \in \prod_{i=1}^{d} V(T_i)$, let

$$E(\boldsymbol{u}) = \{\boldsymbol{x} = (x_1, \ldots, x_d) \in E(G) \mid \forall\, 1 \leq i \leq d : x_i \in \mathrm{desc}(u_i)\}$$

be the set of hyperedges between descendants of $\boldsymbol{u}$'s elements; see Fig. 1 for an example of a two-dimensional tree cross product. We call a tuple $\boldsymbol{u}$ an *induced hyperedge* if $E(\boldsymbol{u}) \neq \emptyset$; the *induced hypergraph* $\mathcal{I}$ consists of the node set $V(\mathcal{I}) = V(G)$ and all induced hyperedges: $E(\mathcal{I}) = \{\boldsymbol{u} \in \prod_{i=1}^{d} V(T_i) \mid E(\boldsymbol{u}) \neq \emptyset\}$. In the example of Fig. 1, for instance, the set $E(v_1, u_2)$ contains three edges, but $E(u_1, u_2)$ is empty; this yields an induced edge $(v_1, u_2) \in E(\mathcal{I})$, but none between u_1 and u_2.

As defined in [3], the *tree cross product problem* is to perform the following operations on tuples $\boldsymbol{u} \in \prod_{i=1}^{d} V(T_i)$:

– `edgeQuery`$(\boldsymbol{u})$ determines if $E(\boldsymbol{u}) \neq \emptyset$, i. e., whether $\boldsymbol{u}$ is induced,
– `edgeReport`$(\boldsymbol{u})$ determines the set $E(\boldsymbol{u})$,
– `edgeExpand`$(\boldsymbol{u}, j)$, where $1 \leq j \leq d$; determines all induced hyperedges $(u_1, \ldots, u_{j-1}, x, u_{j+1}, \ldots, u_d) \in E(\mathcal{I})$, where $x \in \text{children}(u_j)$.

Besides inserting and deleting hyperedges (`newEdge`$(\boldsymbol{u})$ and `deleteEdge`$(\boldsymbol{u})$), our data structure supports the following new operations for adding and removing leaves:

– `addLeaf`(u), where $u \in V(T_j)$; adds a new leaf to T_j as child of node u,
– `deleteLeaf`(u), where u is a leaf in $V(T_j)$ and u is not the root of T_j; removes u and all hyperedges incident to it from T_j.

2.1 The Two-Dimensional Case

With [3] we share the idea to order the nodes of each tree T_1 and T_2 linearly such that for each tree node v the set $\text{desc}(v)$ is fully determined by $\min(v)$ and $\max(v)$, the smallest and the largest node in the subtree rooted at v. Clearly, traversing each tree in post-order yields such an order.

Since `addLeaf`(u) was defined to insert the new leaf as a child of u, insertions occur at any point in the linear order. Therefore, simply assigning consecutive integers to the nodes, as in [3], is inefficient, because all nodes following the new one have to be renumbered. There are already efficient data structures for the problem of performing `order` queries on an ordered list that is subject to `insert` and `delete` operations [6,7]. All of them assign a numerical label (often integers) to the elements, making the `order` query a simple comparison of the labels. The most efficient solution allows all operations to be performed in $\mathcal{O}(1)$ *worst-case* time [7, Sect. 3]. Using a technique for insertions and deletions in dense sequential files [8] on the top level of a four-level data structure, it is, however, rather complicated; a simpler, slightly less efficient, data structure might be more suitable for an implementation. For example, [6] and [7, Sect. 2] both describe solutions with $\mathcal{O}(1)$ *amortized* time for `insert` and `delete` and $\mathcal{O}(1)$ worst-case time for `order`. Since our approach will treat the order maintenance component as a black box, any will do, yet only with an $\mathcal{O}(1)$ worst-case solution the time bounds of Theorems 1 and 2 are worst-case. Besides, we need access to the successor and predecessor of a node in this order. If not already part of the order maintenance structures, we separately maintain a doubly linked list of the nodes for each tree. In the following, we use $<$, $\leq$, $\geq$, and $>$ to compare two nodes instead of the corresponding `order` query.

We keep the list of children at a tree node sorted according to the respective order for the tree. In addition to the values $\min(u_1)$ and $\max(u_1)$, we store at each tree node $u_1 \in V(T_1)$ (and symmetrically for the nodes $u_2 \in V(T_2)$) the set

$$S(u_1) = \{u_2 \in V(T_2) \mid \exists (u_1', u_2) \in E(G) : u_1' \in \mathrm{desc}(u_1)\},$$

i. e., all nodes of the tree T_2 that are connected to a node in the subtree of u_1; see Fig. 1. Although defined as a set, $S(\cdot)$ intrinsically is a multiset: it stores multiple entries of the same node, because the entries correspond to edges, and an `edgeReport` needs to find all of them. A node $u_2' \in S(u_1) \cap \mathrm{desc}(u_2)$ indicates an edge $(u_1', u_2') \in E(G)$ for some $u_1' \in \mathrm{desc}(u_1)$; if no such node exists, no edge connects $\mathrm{desc}(u_1)$ and $\mathrm{desc}(u_2)$. Therefore, `edgeQuery`(u_1, u_2) can be implemented by checking whether $S(u_1) \cap \mathrm{desc}(u_2) = \emptyset$.

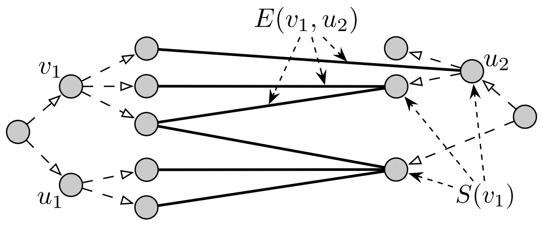

Fig. 1. Example of a two-dimensional tree cross product. The dashed edges belong to the two trees; the solid ones are the hyperedges $E(G)$.

For the sets $S(\cdot)$ we need a data structure that, besides `insert` and `delete`, efficiently supports the *successor* operation `succ`: for $u_1 \in V(T_1)$ and $u_2 \in V(T_2)$, `succ`$(S(u_1), u_2)$ returns the smallest $v \in S(u_1)$ with $v \geq u_2$ or `null` if no such element exists. Observe that $S(u_1) \cap \mathrm{desc}(u_2) \neq \emptyset$ if and only if `succ`$(S(u_1), \min(u_2)) \leq \max(u_2)$; thus, `edgeQuery`$(u_1, u_2)$ returns true if and only if `succ`$(S(u_1), \min(u_2)) \leq \max(u_2)$ [3]. We maintain each set $S(\cdot)$ as a balanced search tree with respect to the order provided by the corresponding order maintenance data structure. Hence, `insert`, `delete`, and `succ` can be done in $\mathcal{O}(\log n)$ worst-case time, provided that the `order` operation is $\mathcal{O}(1)$ worst-case time. Additionally, the leaves of the search trees are linked to facilitate the `edgeReport` operation.

Remark 1. Instead of balanced search trees, in [3] contracted stratified trees (CST) [9] are used for the sets $S(\cdot)$. These improve the time bounds for `insert`, `delete`, and `succ` from $\mathcal{O}(\log n)$ to $\mathcal{O}(\log \log n)$ and increase the required space by a factor of $\mathcal{O}(\log \log n)$; both effects can be seen in Table 1. A CST, however, stores a subset of a *fixed* integer universe; this is impractical here, because the sets $S(\cdot)$ are subsets of the set of tree nodes, which is – in contrast to [3] –

Table 1. Summary of results and comparison for the d-dimensional data structure; all bounds are worst-case. Let $D = \max_{i=1}^{d} \operatorname{depth}(T_i)$. For `edgeReport` and `edgeExpand`, k denotes the size of the output. The `edgeQuery` and `edgeReport` bounds stated in [3] do not include the additive d terms, but they seem unavoidable given the description.

	Approach of [3]	Our data structure
Space	$\mathcal{O}(m(2D)^{d-1} \log\log n)$	$\mathcal{O}(m(2D)^{d-1})$
`edgeQuery`$(\boldsymbol{u})$	$\mathcal{O}(d + \log\log n)$	$\mathcal{O}(d + \log n)$
`edgeReport`$(\boldsymbol{u})$	$\mathcal{O}(\log\log n + k)$	$\mathcal{O}(\log n + k)$
`edgeExpand`$(\boldsymbol{u}, j)$	$\mathcal{O}(d + k \log\log n)$	$\mathcal{O}(d + k \log n)$
`newEdge`$(\boldsymbol{u})$	$\mathcal{O}((2D)^{d-1} \log\log n)$	$\mathcal{O}((2D)^{d-1} \log n)$
`deleteEdge`$(\boldsymbol{u})$	$\mathcal{O}((2D)^{d-1} \log\log n)$	$\mathcal{O}((2D)^{d-1} \log n)$
`addLeaf`(u)	n/a	$\mathcal{O}(D)$
`deleteLeaf`(u)	n/a	$\mathcal{O}(D)$

dynamic. Apart from the universe not being fixed, using the integer labels of the order maintenance structures [6,7] in a CST is complicated: during an `insert` operation, nodes following the inserted one are possibly shifted, i. e., get a new number. Hence, all sets $S(\cdot)$ containing a shifted node need to be updated.

In order to efficiently perform the `edgeExpand` operation (see Algorithm 1), we need to determine the ancestor of a node at a given depth of the tree. This *level ancestor problem* is well studied both in the static [10,11] and the dynamic variant [12,13]. Since we need to add and remove leaves, we will use the dynamic data structure described in [12]. It preprocesses a tree in linear time and space such that level ancestor queries and adding leaves can be performed in $\mathcal{O}(1)$ worst-case time [12, Theorem 6]. Deleting leaves is not explicitly mentioned in [12], but it is obviously possible in constant time by simply deleting the leaf. The space bound, however, would no longer be linear in the number of tree nodes. Similar to maintaining dynamic arrays, we can rebuild the data structure when, for instance, half of the nodes have been deleted. This takes $\mathcal{O}(1)$ amortized cost per delete, but using the standard doubling technique we can distribute it over a sequence of operations such that every operation is worst-case $\mathcal{O}(1)$.

Altogether, our approach combines the idea of [3] with our novel technique of using search trees superimposed over order maintenance structures. This makes the previous data structure more dynamic in regard to insertion and deletion of leaves, while the slow-down for the other operations – roughly a factor of $\mathcal{O}(\log n/\log\log n)$ – is tolerable; see Table 1.

Lemma 1. *Our data structure, as described above, takes $\mathcal{O}(mD \log n)$ worst-case preprocessing time and uses $\mathcal{O}(mD)$ additional space.*

Proof. Each edge $e = (u_1, u_2) \in E(G)$ can contribute an entry only to those sets $S(w)$ where w is an ancestor of either u_1 or u_2. Therefore, the space needed for all sets $S(\cdot)$ together is $\mathcal{O}(mD)$. They are built as follows: for each edge

$e = (u_1', u_2') \in E(G)$, u_1' is inserted into all sets $S(u_2)$, where u_2 is an ancestor of u_2' (and symmetrically u_2' into $S(u_1)$ for each ancestor u_1 of u_1'). These are $\mathcal{O}(mD)$ insert operations in balanced search trees, each of which takes $\mathcal{O}(\log n)$.

The additional space for any of the order maintenance data structures [6,7] is linear in the number of elements they contain, i. e., $\mathcal{O}(n)$; preprocessing takes $\mathcal{O}(n)$ worst-case time. The level ancestor structure also can be preprocessed in linear time and space [12, Theorem 6]. $\square$

Lemma 2. `edgeQuery` *and* `edgeReport` *take* $\mathcal{O}(\log n)$ *and* $\mathcal{O}(\log n + k)$ *worst-case time respectively, where* k *is the number of edges reported.*

Proof. `edgeQuery`(u_1, u_2) is done by checking whether `succ`$(S(u_1), \min(u_2)) \leq \max(u_2)$. Since the set $S(\cdot)$ is stored as a balanced search tree, the `succ` operation and thus the whole `edgeQuery` take $\mathcal{O}(\log n)$ worst-case time. Finding the first edge reported by the `edgeReport` is essentially an `edgeQuery`. Since the leaves of the search trees for the sets $S(\cdot)$ are linked, the remaining edges in $E(u_1, u_2)$ are discovered in constant time each. $\square$

We can implement `edgeExpand` by an appropriate collection of `edgeQuery` operations; in general, however, this is less efficient than Algorithm 1, which is similar to [3]. Since it simplifies the description, Algorithm 1 treats the expansion of the first element of an edge only, i. e., `edgeExpand`$((u_1, u_2), 1)$; expanding the second element works analogously.

Algorithm 1: `edgeExpand`$((u_1, u_2), 1)$

```
input  : $(u_1, u_2) \in E(\mathcal{I})$, i. e., an induced edge $(u_1, u_2)$
output: all children $u_1'$ of $u_1$ such that $(u_1', u_2) \in E(\mathcal{I})$

let $v_1, \dots, v_k$ be the ordered list of children of $u_1$
$R \leftarrow \emptyset$, $t \leftarrow v_1$
repeat
    $s \leftarrow$ succ$(S(u_2), \min(t))$
    if $s \leq \max(v_k)$ then
        if $s > \max(t)$ then  set $t$ to the ancestor of $s$ on the level of children$(u_1)$
        $R \leftarrow R \cup \{t\}$
        if $t \neq v_k$ then  advance $t$ to the next child
    end
until $t = v_k$ or $s > \max(v_k)$
return $R$
```

Lemma 3. `edgeExpand`$((u_1, u_2), j)$ *takes* $\mathcal{O}(k \log n)$ *worst-case time, where* k *is the number of edges reported.*

Proof. Without loss of generality, we assume $j = 1$; the case $j = 2$ is symmetric. Let $v_1, v_2, \dots, v_k$ denote the children of u_1, in ascending order according to the linear order of the tree T_1, i. e., $v_1 < v_2 < \cdots < v_k$. Note that the children are stored in this order and do not need to be sorted. We start with v_1 and determine whether it is connected to u_2 by calculating $s =$ `succ`$(S(u_2), \min(v_1))$. If $s \leq \max(v_1)$, v_1 is reported; if $\max(v_1) < s \leq \max(v_k)$, the ancestor s'

of s among the children of u_1 is reported by way of the level ancestor data structure. This procedure is iterated until $s > \max(v_k)$; see Algorithm 1. Each `succ` operation, except the last, yields a new result. Since determining the level ancestor takes constant time, we get $\mathcal{O}(k \log n)$ worst-case time. $\square$

After adding a new edge (u_1', u_2') to $E(G)$, we insert u_1' into $S(u_2)$ for all ancestors u_2 of u_2' and u_2' into $S(u_1)$ for the ancestors u_1 of u_1'; conversely, for deleting an edge (u_1', u_2') we remove u_1' from $S(u_2)$ and u_2' from $S(u_1)$. Since there are at most $2D$ ancestors, this yields the following lemma:

Lemma 4. `newEdge`(u_1, u_2) *and* `deleteEdge`(u_1, u_2) *take* $\mathcal{O}(D \log n)$ *time each.*

For deleting a leaf u, we first delete all incident edges with `deleteEdge`, which implicitly updates all affected sets $S(\cdot)$. Next, we update the order maintenance and the level ancestor data structures and remove the leaf from its tree. At each ancestor u' of u we possibly have to update the values $\min(u')$ and $\max(u')$ to u's predecessor or successor in the ordered list of tree nodes.

When inserting a new leaf u' as a child of node u, we first insert u' right before $\max(u)$ into the order maintenance structure. Then we add u' to the level ancestor data structure and insert it into the tree as a child of u. If u was an inner node before this operation, the values $\min(u)$ and $\max(u)$ remain correct. But if u was a leaf, i. e., $\max(u) = \min(u)$, we have to set $\min(u) = u'$; this may cause further updates of the $\min(\cdot)$ values at ancestors of u. Since insertion and deletion in the order maintenance as well as the level ancestor data structure take constant time, this yields the following lemma:

Lemma 5. *Deleting a leaf (without any incident edges) and inserting a leaf can be performed in* $\mathcal{O}(D)$ *time.*

2.2 Higher Dimensions

The data structure described so far maintains a dynamic set of pairs $(u_1, u_2) \in V(T_1) \times V(T_2)$, while both trees are dynamic in regard to insertion and deletion of leaves. It provides the retrieval operations `edgeQuery`, `edgeReport`, and `edgeExpand`. We will give a recursive description for the higher dimensional data structure with the two-dimensional case as basis.

Suppose that there is already such a data structure for the case d, i. e., for maintaining hyperedges between nodes of d trees. The $(d+1)$-dimensional data structure stores at each node $u_1 \in V(T_1)$ the set

$$S_{d+1}(u_1) = \{(u_1', u_2', \dots, u_{d+1}') \in E(G) \mid u_1' \in \mathrm{desc}(u_1)\},$$

i. e., all hyperedges incident with descendants of u_1. Disregarding the first element of the hyperedges, we use a separate d-dimensional data structure for each set $S_{d+1}(\cdot)$. In other words, we store the $(d+1)$-dimensional hyperedges in a d-dimensional data structure according to their projections onto the last d elements.

We can implement `edgeQuery`$(u_1, u_2, \ldots, u_{d+1})$ as `edgeQuery`$(u_2, \ldots, u_{d+1})$ on the d-dimensional data structure stored at u_1. An `edgeReport` query is forwarded similarly; the edges it returns are already the correct $(d+1)$-dimensional result, because the d-dimensional data structure contains the original hyperedges. The operation `edgeExpand`$((u_1, u_2, \ldots, u_{d+1}), j)$ for $j \neq 1$ is implemented as an `edgeExpand`$((u_2, \ldots, u_{d+1}), j)$ on the d-dimensional data structure at the node u_1. For expanding a hyperedge at its first element ($j = 1$), we build the same data structure designating some other tree to be T_1, for instance T_2.

Theorem 1. *With $\mathcal{O}(m(2D)^{d-1})$ additional space, our data structure solves the d-dimensional dynamic tree cross product problem with the worst-case time bounds shown in Table 1.*

Proof. For $d = 2$, all bounds in Table 1 follow directly from Lemmas 1, 2, 3, 4, and 5.

For $d > 2$, an edge $(u_1, \ldots, u_d)$ contributes an entry to the lower dimensional data structure at each ancestor of u_1 and at each ancestor of u_2 (assuming that T_2 is the tree designated to be T_1 for the second data structure). These are $\mathcal{O}(2D)$ entries; by induction each entry uses $\mathcal{O}((2D)^{(d-1)-1})$ space in the lower dimensional data structure, which gives a total of $\mathcal{O}(m(2D)^{d-1})$ space. Inserting or deleting an edge is implemented as one insert or delete operation in a lower dimensional data structure for each ancestor in the two dimensions, each of which takes $\mathcal{O}((2D)^{(d-1)-1} \log n)$ by induction. All retrieval operations `edgeQuery`, `edgeReport`, and `edgeExpand` are forwarded to an appropriate lower dimensional data structure; this recursion ends at some two dimensional data structure, where the operation is implemented as described in Sect. 2.1. Hence, we get additional $d - 1$ steps for the recursion. Inserting and deleting a leaf is exactly the same as in the two dimensional case. □

Remark 2. In [3] *compressed trees* [14] are used to improve the space bound. For a tree T, an edge $(\text{parent}(u), u)$ is *light* if $2|\text{desc}(u)| \leq |\text{desc}(\text{parent}(u))|$ and *heavy* otherwise. The compressed tree $C(T)$ evolves from T by contracting all heavy paths into their respective topmost node. This technique could be employed here as well, but maintaining $C(T)$ subject to insertion and deletion of leaves into the original tree T is not straightforward. The problem is that these modifications can change the status of tree edges at ancestors of the affected node from light to heavy and vice versa. In the compressed tree this results in adding or removing an inner node. Especially for a new inner node this is expensive, because we have to equip the new node with appropriate data structures, e. g., the set $S(\cdot)$. While reducing the space bound, using compressed trees increases the time bounds of most operations by a factor of $\mathcal{O}(\log n/(\log\log n)^2)$. In [3] the trees are stratified recursively to improve these time bounds again. Unfortunately, the stratification arguments are faulty [15]; therefore, only the results without stratification are listed in Table 1.

3 Maintaining Hierarchical Graph Views

A *compound graph* $\varGamma = (V, E_i, E_a)$ [4] consists of nodes V, *inclusion edges* E_i, and *adjacency edges* E_a. It is required that the *inclusion digraph* $T = (V, E_i)$ is a tree and no adjacency edge connects a node to one of its descendants or ancestors; see Figs. 2 and 3. A *view* U is a graph with nodes $V(U) \subseteq V$ such that $\forall\, u, v \in V(U) : \mathrm{desc}(u) \cap \mathrm{desc}(v) = \emptyset$, i. e., the nodes of the view are not related in terms of the inclusion tree. Two nodes $u, v \in V(U)$ are connected by an *induced edge* if and only if there are nodes $u' \in \mathrm{desc}(u)$ and $v' \in \mathrm{desc}(v)$ such that u' and v' are connected by an adjacency edge $(u', v') \in E_a$; see Fig. 4.

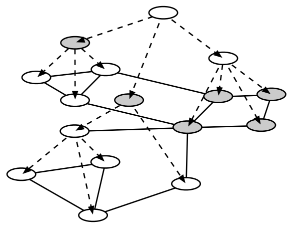
Fig. 2. An example of a compound graph: the directed edges form the inclusion digraph T; the undirected ones are the adjacency edges E_a

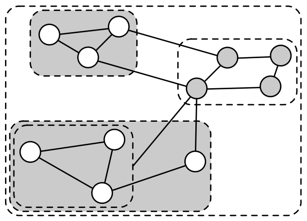
Fig. 3. The same compound graph as in Fig. 2, but T is depicted by the inclusion of the dashed rectangles

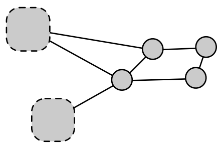
Fig. 4. The view consisting of the darker shaded nodes of the compound graph in Figs. 2 and 3

Given a compound graph $\varGamma$ and a view U, the *graph view maintenance problem*, according to [2], is to efficiently perform the following operations on U:

- `expand`(v), where $v \in V(U)$; replaces node v with its children, i. e., the result is the view U' with nodes $V(U') = V(U) \setminus \{v\} \cup \mathrm{children}(v)$,
- `contract`(v), where $\mathrm{children}(v) \subseteq V(U)$; contracts all children of v, i. e., the result is the view U' with nodes $V(U') = V(U) \setminus \mathrm{children}(v) \cup \{v\}$.

In the new dynamic leaves variant of this problem, the compound graph $\varGamma$ is subject to the following modifications:

- `newEdge`(u, v), where $u, v \in V$, $u \notin \mathrm{desc}(v)$, and $v \notin \mathrm{desc}(u)$; adds a new adjacency edge (u, v) to $\varGamma$,
- `deleteEdge`(u, v), where $(u, v) \in E_a$; removes adjacency edge (u, v) from $\varGamma$,
- `newLeaf`(u), where $u \in V$; adds a new node v to $\varGamma$ and a new inclusion edge (u, v), i. e., v becomes a child of u in the inclusion tree,
- `deleteLeaf`(u), where u is a leaf in the inclusion tree; removes u from G.

Table 2. Results and comparison for the graph view maintenance problem. Let $D = \text{depth}(T)$, $s = \min\{D, \log n\}$, and $\text{Opt}(U, v) = \sum_{v' \in \text{children}(v)} |\text{adj}_U(v')|$. For `expand`$(v)$, U' denotes the view after expanding v in U. The bounds labeled with $^{\text{exp}}$ are expected, all others are worst-case.

	Approach of [2]	Approach of [3]	Our data structure
Space	$\mathcal{O}(ms^2)$	$\mathcal{O}(mD \log\log n)$	$\mathcal{O}(mD)$
`expand`(v)	$\mathcal{O}(\text{Opt}(U', v))$	$\mathcal{O}(\text{Opt}(U', v) \log\log n)$	$\mathcal{O}(\text{Opt}(U', v) \log n)$
`contract`(v)	$\mathcal{O}(\text{Opt}(U, v))$	$\mathcal{O}(\text{Opt}(U, v))$	$\mathcal{O}(\text{Opt}(U, v))$
`newEdge`(u, v)	$\mathcal{O}^{\text{exp}}(s^2 \log n)$	$\mathcal{O}(D \log\log n)$	$\mathcal{O}(D \log n)$
`deleteEdge`(u, v)	$\mathcal{O}^{\text{exp}}(s^2 \log n)$	$\mathcal{O}(D \log\log n)$	$\mathcal{O}(D \log n)$
`newLeaf`(u)	n/a	n/a	$\mathcal{O}(D)$
`deleteLeaf`(u)	n/a	n/a	$\mathcal{O}(D)$

Besides compound graphs, there are other concepts for extending graphs with a hierarchical structure [5,16,17,18], among which *clustered graphs* [5] are very popular; they consist of a base graph and a tree whose leaves are exactly the nodes of the base graph. Consequently, clustered graphs have adjacency edges only between leaves of the inclusion tree, whereas *compound graphs* allow them between any pair of tree nodes such that neither is a descendant of the other. In [2,3] data structures for maintaining views of clustered graphs under `expand` and `contract` operations are described; these are either static or allow insertion and deletion of adjacency edges. Efficient data structures that additionally support modifications of the node set were left as an open problem [2]. Providing insertion and deletion of leaves, we solve this problem partially; see also [19] for a detailed description on directly applying the ideas of Sect. 2.1 to this problem. Table 2 summarizes our results and compares them to the other approaches.

The graph view maintenance problem for compound graphs can be reduced to a two-dimensional tree cross product; see [3, Sect. 5.1]. We set $T_1 = T_2 = T$ and interpret an adjacency edge $(u, v) \in E_a$ as an edge connecting $u \in T_1$ and $v \in T_2$. Clearly, determining whether there is an induced edge between two nodes u and v becomes an `edgeQuery`; `newEdge` and `deleteEdge` directly map to corresponding operations for tree cross products. Inserting or deleting leaves in the inclusion tree engenders a `newLeaf` or `deleteLeaf` operation on both T_1 and T_2.

For expanding a view U at the node v, we use an `expandEdge`$((v, w), 1)$ for each edge (v, w) incident to v in the view U; this determines all the children of v inheriting the edge (v, w). Contracting a view at the node v is straightforward: all the children of v are removed and v is connected to all former neighbors of children of v.

Theorem 2. *Let $D = \text{depth}(T)$; with $\mathcal{O}(mD)$ additional space, our data structure solves the dynamic leaves variant of the graph view maintenance problem with the worst-case time bounds shown in Table 2.*

Proof. As in [2,3], let $\text{Opt}(U, v) = \sum_{v' \in \text{children}(v)} |\text{adj}_U(v')|$, where $\text{adj}_U(v')$ are the edges incident to v' in the view U. The number of items that have to be added or removed during `expand`(v) is bounded by $\mathcal{O}(\text{Opt}(U', v))$, where U' is the view after expanding v in U. Similarly, the number of items affected by a `contract`(v) is bounded by $\mathcal{O}(\text{Opt}(U, v))$. By traversing all edges incident to children of v, we can find the neighbors of v in U', where U' is the view resulting from contracting the children of v in U. Hence, `contract`(v) takes $\mathcal{O}(\text{Opt}(U, v))$ time. `expand`(v) is bounded by $\mathcal{O}(\text{Opt}(U', v) \log n)$, for we have to expand each edge incident to v in U; see Algorithm 1. Note that this does not yield the edges between two children of v: we maintain these edges separately in a list at every node of the tree. Since each edge is stored exactly once in such a list, namely at the nearest common ancestor of its end nodes, this uses $\mathcal{O}(m)$ additional space, which does not violate the $\mathcal{O}(mD)$ space bound. Clearly, it takes additional $\mathcal{O}(D)$ time for updating these lists, which is possible within the $\mathcal{O}(D \log n)$ bound for inserting and deleting edges. All other bounds follow immediately from Theorem 1. □

4 Conclusion

We have presented an efficient data structure for range searching over tree cross products, where the trees are dynamic with regard to insertion and deletion of leaves. As summarized in Table 1, our approach can compete well with the one it extends [3]. So far, it is the only data structure for tree cross products where the node set is dynamic. Applying it to graph view maintenance, we have partially solved the dynamic graph and tree variant, an open problem in [2]. The comparison in Table 2 shows that our solution matches with the more static ones, but additionally provides insertion and deletion of graph nodes. A data structure for the dynamic graph and tree variant, i. e., with splitting and merging of clusters, remains an open problem.

Acknowledgments

I would like to thank Adam Buchsbaum for the enlightening discussions on the details of [3] and for his valuable comments on drafts of this paper. Furthermore, I am grateful to Franz Brandenburg, Christian Bachmaier, and Falk Schreiber for their suggestions.

References

1. Brandenburg, F.J., Forster, M., Pick, A., Raitner, M., Schreiber, F.: Biopath – exploration and visualization of biochemical pathways. In Mutzel, P., Jünger, M., eds.: Graph Drawing Software. Mathematics and Visualization. Springer (2003) 215–236
2. Buchsbaum, A.L., Westbrook, J.R.: Maintaining hierarchical graph views. In: Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA). (2000) 566–575

3. Buchsbaum, A.L., Goodrich, M.T., Westbrook, J.R.: Range searching over tree cross products. In Paterson, M., ed.: Proc. 8th European Symposium on Algorithms (ESA). Volume 1879 of LNCS. (2000) 120–131
4. Sugiyama, K., Misue, K.: Visualization of structural information: Automatic drawing of compound digraphs. IEEE Transactions on Systems, Man, and Cybernetics **21** (1991) 876–892
5. Feng, Q.W., Cohen, R.F., Eades, P.: How to draw a planar clustered graph. In Du, D.Z., Li, M., eds.: Proc. 1st Intl. Conference on Computing and Combinatorics (COCOON). Volume 959 of LNCS. (1995) 21–30
6. Bender, M.A., Richard, C., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In Möhring, R.H., Raman, R., eds.: Proc. 10th European Symposium on Algorithms (ESA). Volume 2461 of LNCS. (2002) 152–164
7. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: 9th ACM Symposium on Theory of Computing (STOC). (1987) 365–372
8. Willard, D.E.: Good worst-case algorithms for inserting and deleting records in dense sequential files. In: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data. (1986) 251–260
9. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory **10** (1977) 99–127
10. Berkman, O., Vishkin, U.: Finding level ancestors in trees. Journal of Computer and System Sciences **48** (1994) 214–230
11. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. In Rajsbaum, S., ed.: Proc. 5th Latin American Symposium on Theoretical Informatics (LATIN). Volume 2286 of LNCS. (2002) 508–515
12. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In Montanari, U., Rolim, J.D.P., Welzl, E., eds.: Automata, Languages and Programming, 27th International Colloquium, ICALP 2000. Volume 1853 of LNCS. (2000) 73–84
13. Dietz, P.F.: Finding level ancestors in dynamic trees. In Dehne, F.K.H.A., Sack, J.R., Santoro, N., eds.: Algorithms and Data Structures, 2nd Workshop WADS '91. Volume 519 of LNCS. (1991) 32–40
14. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing **13** (1984) 338–355
15. Buchsbaum, A.L.: Personal communication
16. Harel, D.: On visual formalisms. Comm. of the ACM **31** (1988) 588–600
17. Lai, W., Eades, P.: A graph model which supports flexible layout functions. Technical Report 96–15, University of Newcastle (1996)
18. Raitner, M.: HGV: A library for hierarchies, graphs, and views. In Goodrich, M.T., Kobourov, S.G., eds.: Proc. 10th Intl. Symposium on Graph Drawing (GD). Volume 1528 of LNCS. (2002) 236–243
19. Raitner, M.: Maintaining hierarchical graph views for dynamic graphs. Technical Report MIP-0403, Universität Passau (2004)