Range Searching Over Tree Cross Products

Adam L. Buchsbaum[1], Michael T. Goodrich[2⋆], and Jeffery R. Westbrook[3⋆⋆]

[1] AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932,
alb@research.att.com, http://www.research.att.com/info/alb
[2] Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218,
goodrich@jhu.edu, http://www.cs.jhu.edu/~goodrich/home.html
[3] 20th Century Television, Los Angeles, CA 90025,
jwestbrook@acm.org

Abstract. We introduce the *tree cross-product problem*, which abstracts a data structure common to applications in graph visualization, string matching, and software analysis. We design solutions with a variety of tradeoffs, yielding improvements and new results for these applications.

1 Introduction

Range searching is a classic problem in data structure design [8, 10, 12, 23, 29]. It is typically defined on a *ground set*, S, of ordered tuples $(x_1, \ldots, x_d)$ that are to be queried by *ranges*, which are specified by intersections of tuples of intervals on the coordinates. A *simple query* asks if such a range R contains any point of S, and a *reporting query* asks for all points of S inside R.

We introduce and study an interesting variant of range searching, which we call *range searching over tree cross products*. Given a priori are d rooted trees, $T_1, \ldots, T_d$, and a ground set E of tuples $(x_1, \ldots, x_d)$, such that $x_i \in T_i$, $1 \leq i \leq d$. The nodes of the trees and the set E define a d-partite hypergraph G. We wish to perform several possible queries on tuples of nodes $\boldsymbol{u} = (u_1, \ldots, u_d)$, such that $u_i \in T_i$, $1 \leq i \leq d$. Analogous to a simple range searching query, an *edge query* determines if there is a hyperedge in G that connects descendents of all the u_i nodes. We say that such a hyperedge *induces* $\boldsymbol{u}$. Likewise, a *reporting query* determines all hyperedges in G that induce $\boldsymbol{u}$. An *expansion query* determines, for each y that is a child of some designated u_i, whether the tuple formed by replacing u_i with y in $\boldsymbol{u}$ is induced by a hyperedge in G. The goal is to preprocess the trees and hypergraph so that these queries can be performed efficiently online. Dynamic variants admit updates to the trees and hypergraph.

1.1 Motivating Applications. We explore four applications of tree cross products. (We define the applications formally in Section 5.) First is the *hierarchical graph-view maintenance problem*, which has applications in graph visualization [7, 9]. We are given a rooted tree, T, and a graph, G, of edges superimposed between leaves of T. At any point, there exists a *view*, U, of G,

⋆ Work supported by NSF Grant CCR-9732300 and ARO Grant DAAH04-96-1-0013.
⋆⋆ Work completed while a member of AT&T Labs.

which is a set of nodes of T such that each leaf of T has precisely one ancestor in U. The *induced graph*, G/U, is obtained by contracting each vertex in G into its representative in U, removing multiple and loop edges. The problem is to maintain G/U as nodes in U are expanded (replaced by their children) or contracted (replace their children). Buchsbaum and Westbrook [7] show how to expand and contract nodes in U in optimal time for an unweighted graph G: linear in the number of changes to G/U. The tree cross-product problem generalizes the graph-view problem. Our approach is orthogonal to that of [7], however.

Two more applications involve string matching. In *text indexing with one error*, we want to preprocess a string T of length n so that we can determine on-line all the occurrences of a pattern P of length m with one error (using Hamming or Levenshtein [19] distance). This problem has applications in password security [20] and text filtering [3, 4, 22, 24]. We improve on the work of Ferragina, Muthukrishnan, and de Berg [11] and Amir et al. [2], whose solution traverses two suffix trees in tandem, periodically performing analogous reporting queries. Grossi and Vitter [15] use a similar strategy to report contiguous entries in compressed suffix arrays, and we also improve their bounds.

Finally, we consider finding *hammocks* in directed graphs. These are regions of nodes that are separated from designated source and sink nodes by the equivalent of articulation points. Hammocks have been studied in the context of compiler control-flow analysis [17], in which solutions to the problem of finding all hammocks off-line are known. We present the first results for finding hammocks on-line, with an application to software system analysis [5].

1.2 Our Results. We contribute a formal definition of tree cross-product operations, and we relate them to range searching. Rather than use classical range search techniques, we exploit the structure of the input to devise a framework based upon simpler search structures. Let $n = \sum_{i=1}^{d} |T_i|$, $m = |E|$, and k be the number of edges reported by a reporting or expansion query. Using $O(m \log^{\frac{d-1}{2}} n)$ (rsp., $O(m(\log\log n)^{d-1})$) space, we can perform edge queries in $O(2^{d-1} \log n / \log\log n)$ (rsp., $O(\log n(\log\log n)^{d-2})$) time; reporting queries add $O(k)$ to the edge-query time, and expansion queries multiply the edge-query time by $O(k)$. In the dynamic case, we can perform insertions and deletions of hyperedges in G in $O(\log^{\frac{d+1}{2}} n / \log\log n)$ (rsp., $O(\log n(\log\log n)^{d-2})$) time. In the two-dimensional case, note that we can achieve logarithmic query **and** update times in almost-linear space. No classical range searching result provides the same bounds. In the static case, the query times improve by $\log n/(\log\log n)^2$ factors, and the preprocessing time equals the space. All are deterministic, worst-case bounds. Our framework allows simple implementations for practical cases, using nothing more sophisticated than balanced search trees.

Applied to graph views, we present a dynamic solution that improves the results of Buchsbaum and Westbrook [7] by factors of $\log^2 n / \log\log n$ in space (to $O(m \log\log n)$) and $\log^2 n$ in update time (to $O(\log n)$). The cost is a $\log n$ factor penalty in expand time; contract remains optimal. For the static problem, the space (and preprocessing time) reduction is the same, while the expand penalty is only $(\log\log n)^2$. All of our bounds are deterministic, worst-case, whereas the

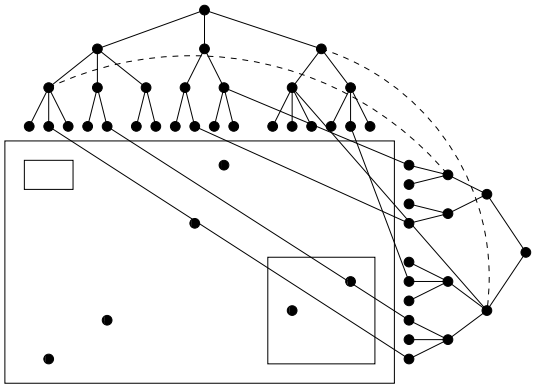

Fig. 1. Tree cross products for $d = 2$. Graph edges are shown as solid arcs and corresponding points; two potential induced edges are shown as dashed arcs and corresponding rectangular ranges.

prior update and preprocessing time bounds [7] are expected. For text indexing with one error, we improve all the bounds of Amir et al. [2] by $\log n$ factors. For compressed suffix-array traversal, we improve the additional reporting time of Grossi and Vitter [15] by a factor of $\log n/(\log\log n)^2$. For on-line hammock finding, we give a data structure that uses $O(n\sqrt{\log n})$ (rsp., $O(n \log\log n)$) space and preprocessing time and returns hammocks of size k in $O(\log\log n + k)$ (rsp., $O((\log\log n)^2 + k)$) time each; these are the first such results.

Table 1 in Section 4 summarizes our results.

2 Tree Cross Products

Let $G = (V(G), E(G))$ denote a (hyper)graph with vertex set $V(G)$ and edge set $E(G)$. Let T be a rooted tree. For $v \in V(T)$: let *children*(v) be the set of children of v, and let *desc*(v) be the set of descendents of v. Given d rooted trees, $T_1, \ldots, T_d$, consider some d-partite hypergraph, G, such that $V(G) = \bigcup_{i=1}^{d} V(T_i)$ and $E(G) \subseteq V(T_1) \times \cdots \times V(T_d)$, which we also write $\prod_{i=1}^{d} V(T_i)$. G is a subset of the *cross product of* $T_1, \ldots, T_d$. Denote by $\boldsymbol{u}$ an element $(u_1, \ldots, u_d) \in \prod_{i=1}^{d} V(T_i)$. Define $E(\boldsymbol{u})$ to be the set $\{\boldsymbol{x} \in E(G) : x_i \in \mathit{desc}(u_i), 1 \le i \le d\}$, i.e., the set of hyperedges of G connecting descendents of each component of $\boldsymbol{u}$.

Let $\mathcal{I}$ be the hypergraph with vertex set $\bigcup_{i=1}^{d} V(T_i)$ and edge set $\{\boldsymbol{u} : |E(\boldsymbol{u})| > 0\}$. We call $\mathcal{I}$ an *induced hypergraph* and hyperedges of $\mathcal{I}$ *induced hyperedges*, because they are induced by the cross-product subset G of the T_is. (See Fig. 1.) If each hyperedge $\boldsymbol{e} \in E(G)$ has a weight, $w(\boldsymbol{e}) \in \mathcal{G}$, from a given abelian group, $(\mathcal{G}, \odot)$, then the *weight* of $\boldsymbol{u} \in E(\mathcal{I})$ is defined to be $w(\boldsymbol{u}) = \bigodot \{w(\boldsymbol{x}) : \boldsymbol{x} \in E(\boldsymbol{u})\}$.

Given $T_1, \ldots, T_d$, and G, the *tree cross-product problem* is to perform on-line the following *tree cross-product operations* on tuples $\boldsymbol{u} \in \prod_{i=1}^{d} V(T_i)$.

edgeQuery$(\boldsymbol{u})$**.** Determine if $\boldsymbol{u} \in E(\mathcal{I})$.

edgeReport$(\boldsymbol{u})$**.** Determine $E(\boldsymbol{u})$.

edgeWeight$(\boldsymbol{u})$**.** Determine $w(\boldsymbol{u})$ (undefined if $\boldsymbol{u} \notin E(\mathcal{I})$).

edgeExpand$(\boldsymbol{u}, j)$**,** where $1 \le j \le d$. For each $x \in \mathit{children}(u_j)$, determine if $(u_1, \ldots, u_{j-1}, x, u_{j+1}, \ldots, u_d) \in E(\mathcal{I})$.

We consider the static version of the problem and also one in which hyperedges can be added and deleted. Although we can implement *edgeExpand*$(\boldsymbol{u}, j)$ as an appropriate collection of *edgeQuery*$(\cdot)$ operations, we demonstrate that considering *edgeExpand*$(\boldsymbol{u}, j)$ as a primitive can lead to more efficient solutions.

We denote by T the forest containing $T_1, \ldots, T_d$, and define $n = |V(G)| = |V(T)|$, and $m = |E(G)|$. For $v \in V(T)$: let *depth*(v) be the depth of v; let $p(v)$ be the parent of v; and let *leaves*(v) be the set of leaf descendents of v. Let *depth*$(T) = \max_{v \in V(T)}$ *depth*(v). We assume T is given explicitly, although our methods extend when it is implicitly defined by *depth*$(\cdot)$, $p(\cdot)$, *children*$(\cdot)$, etc.

In an $O(n)$-time preprocessing phase, we perform postorder traversals of $T_1, \ldots, T_d$ so that each node is assigned its postorder number (ordinally from 1 for each tree) and depth. We abuse notation and use u itself to denote the postorder number of node u in its respective tree. For any node $u \in V(T)$, define $\min(u) = \min\{x : x \in \textit{desc}(u)\}$ and $\max(u) = \max\{x : x \in \textit{desc}(u)\}$.

3 A Range Searching Framework

We first assume $d = 2$ (and discuss G in terms of a graph with edges of the form (u, v)) and later extend our solutions to higher dimensions. Determining if $(u, v) \in E(\mathcal{I})$ can be viewed as a two-dimensional range query. Consider a $|V(T_1)| \times |V(T_2)|$ integral grid, with a point (x, y) for each edge $(x, y) \in E(G)$, as shown in Fig. 1. $E(u, v)$ corresponds precisely to the points in the range $([\min(u), \max(u)], [\min(v), \max(v)])$. A straightforward solution applies classical range searching results, but this ignores the structure imposed by T.

Each of T_1 and T_2 defines only $O(n)$ one-dimensional subranges that can participate in any two-dimensional query. To exploit this structure, we store a set $S(u)$ with each node $u \in V(T)$, which contains the far endpoints of all graphs edges incident on vertices in *desc*(u). For $u \in V(T)$, we maintain the invariant that $S(u) = \{y : (x, y) \in E(G) \vee (y, x) \in E(G),\ x \in \textit{desc}(u)\}$. Each $y \in S(u)$ is thus in the tree other than that containing u. The operations on $S(u)$ are *insert*$(S(u), y)$, *delete*$(S(u), y)$, and *succ*$(S(u), y)$. Operation *succ*$(S(u), y)$ returns the successor of y in $S(u)$, or ∞ if there is none.

Thus, (u, v) is an induced edge of $\mathcal{I}$ if and only if (1) u and v are in separate trees, and (2) *succ*$(S(u), \min(v) - 1) \leq \max(v)$. By the invariant that defines $S(u)$, test (2) succeeds if there is an edge from a descendent of u to one of v. Equivalently, test (2) can be replaced by (2′) *succ*$(S(v), \min(u) - 1) \leq \max(u)$.

To implement *edgeExpand*$((u, v), u)$ (sym., *edgeExpand*$((u, v), v)$), we iteratively perform *succ*$(S(v), \min(x) - 1)$ on the children x of u, in left-to-right order, using the intermediate results to skip children with no induced edges. This is more efficient than performing *edgeQuery*(x, v) for each $x \in$ *children*(u).

To insert edge (x, y) into $E(G)$, we perform *insert*$(S(u), y)$ for each node u on the x-to-*root*(T_1) path and *insert*$(S(v), x)$ for each node v on the y-to-*root*(T_2) path. Deletion of (x, y) substitutes *delete*$(\cdot, \cdot)$ for *insert*$(\cdot, \cdot)$.

Theorem 1. *Let $D =$ depth(T). With $O(mD \log\log n)$ space, we can insert or delete an edge into G in $O(D \log\log n)$ time and perform edgeQuery$(\cdot)$ in*

$O(\log\log n)$ time, edgeExpand$(\cdot,\cdot)$ in $O(k\log\log n)$ time, and edgeReport$(\cdot)$ in $O(\log\log n + k)$ time, where k is the number of edges reported.

Proof (Sketch): We maintain each set $S(u)$ as a contracted stratified tree (CST) [25], linking the leaves to facilitate *edgeReport*$(\cdot)$. Each edge in $E(G)$ appears in at most $2D$ such sets. The number of *succ*$(\cdot,\cdot)$ operations for *edgeExpand*$(\cdot,\cdot)$ is one plus the number of induced edges returned, because each *succ*$(\cdot,\cdot)$ operation except the last returns a distinct induced edge. □

Ferragina, Muthukrishnan, and de Berg [11] similarly solve the related *point enclosure problem* on a multi-dimensional grid. They maintain a recursive search structure on the grid, however, whereas we exploit the structure of T.

4 Decompositions and Higher Dimensions

4.1 Compressed Trees and Three-Sided Range Queries. When $D = \omega(\log n/\log\log n)$, we can improve the space bound using compressed trees [13, 16, 27]. Call tree edge $(v, p(v))$ *light* if $2|desc(v)| \le |desc(p(v))|$, and *heavy* otherwise. Each node has at most one heavy edge to a child, so deletion of the light edges produces a collection of node-disjoint *heavy paths*.

The *compressed forest*, $C(T)$, is constructed (in $O(n)$ time) by contracting each heavy path in T into a single node. Each tree edge in $C(T)$ corresponds to a light edge of T. Since there are $O(\log n)$ light edges on the path from any node to the root of T, $C(T)$ has depth $O(\log n)$. Let $h(\nu)$, $\nu \in C(T)$, denote the heavy path of T that generates node ν. Define $h^{-1}(v) = \nu$ for all $v \in h(\nu)$.

Consider node $u \in T$ and the corresponding node $\mu = h^{-1}(u) \in C(T)$. Number the nodes in the heavy path $h(\mu)$ top-down $(u_1, \ldots, u_\ell)$. For some $1 \le i \le \ell$, $u = u_i$. Associated with u are the corresponding node $\mu \in C(T)$, the value $index(u) = i$ and a pointer, $t(u)$, to u_1 (which is a node in T). We also define $t(\mu) = u_1$, the top of the heavy path $h(\mu)$. For a node $\mu \in C(T)$ and vertex $x \in desc(t(\mu))$, we define $entry(\mu, x)$ to be the maximum i such that $x \in desc(u_i)$, where u_i is the ith node in $h(\mu)$.

We now maintain the sets $S(\cdot)$ on nodes in $C(T)$, not T. For $\mu \in V(C(T))$, $S(\mu) = \{(y, entry(\mu, x)) : (x, y) \in E(G) \vee (y, x) \in E(G),\ x \in desc(t(\mu))\}$. Consider $u \in V(T_1)$, $v \in V(T_2)$, $\mu = h^{-1}(u)$, and $\nu = h^{-1}(v)$. (u, v) is an induced edge of $\mathcal{I}$ if and only if there exists some $(y, d) \in S(\mu)$ such that (a) $\min(v) \le y \le \max(v)$ and (b) $index(u) \le d$. (a) implies that y is a descendent of v; (b) implies that y is adjacent to a descendent of a node at least as deep as u on $h(\mu)$ and thus to a descendent of u. We can also query $S(\nu)$ symmetrically.

The update operations become $insert(S(\mu), (i, j))$ and $delete(S(\mu), (i, j))$. The query operations are: $tsrQuery(S(\mu), (x_1, x_2), y)$, which returns an arbitrary pair $(i, j) \in S(\mu)$ such that $x_1 \le i \le x_2$ and $j \ge y$, or $\emptyset$ if none exists; and $tsrReport(S(\mu), (x_1, x_2), y)$, which returns the set of such pairs. An $edgeQuery(u, v)$ is effected by $tsrQuery(S(\mu), (\min(v), \max(v)), index(u))$, and an $edgeReport(u, v)$ by $tsrReport(S(\mu), (\min(v), \max(v)), index(u))$ (or symmetrically on $S(\nu)$). These queries are sometimes called *three-sided range queries*.

We implement *edgeExpand*$((u,v),\cdot)$ iteratively, as in Section 3. To update $E(G)$, we use the $t(\cdot)$ values to navigate up T and the $h^{-1}(\cdot)$ and *index*$(\cdot)$ values to create the proper arguments to *insert*$(\cdot,\cdot)$ and *delete*$(\cdot,\cdot)$.

Theorem 2. *Let $p = \min\{\mathit{depth}(T), \log n\}$. With $O(mp)$ space, we can insert or delete an edge into G in $O(p \log n / \log\log n)$ time and perform edgeQuery$(\cdot)$ in $O(\log n / \log\log n)$ time, edgeExpand$(\cdot,\cdot)$ in $O(k \log n / \log\log n)$ time, and edgeReport$(\cdot)$ in $O(\log n / \log\log n + k)$ time; k is the number of edges reported.*

Proof (Sketch): We maintain each $S(\mu)$ as a separate priority search tree (PST) [29]. Each edge $(x,y) \in E(G)$ appears in at most $2p$ such sets. During an *edgeExpand*$(\cdot,\cdot)$, each *tsrQuery*$(\cdot,\cdot,\cdot)$ either engenders a new induced edge or else terminates the procedure □

Theorem 3. *Let $p = \min\{\mathit{depth}(T), \log n\}$. With $O(mp)$ preprocessing time and space, we can build a data structure that performs edgeQuery$(\cdot)$ in $O(\log\log n)$ time, edgeExpand$(\cdot,\cdot)$ in $O(k \log\log n)$ time, and edgeReport$(\cdot)$ in $O(\log\log n + k)$ time, where k is the number of edges reported.*

Proof (Sketch): We use a static three-sided range query data structure [12, 23]. To provide access to the leaves of the underlying data structures without the high overhead (e.g., perfect hashing) of previous solutions [12, 23], we use one array of size n. Each element i points to a CST that contains pointers to the leaf representing i in each structure, indexed by structure, which we number ordinally 1 to $|V(C(T))|$. Each leaf in each underlying structure appears in one such CST, so the total extra space and preprocessing time is $O(n + m \log\log n)$. The initial access to a leaf requires an $O(\log\log n)$-time CST look-up. □

4.2 Stratification. We further reduce the space by stratifying T into $\sqrt{D}$ strata of $\sqrt{D}$ levels each, where $D = \mathit{depth}(T)$. Entries for an edge (u,v) are made in set $S(x)$ only for each x that is in ancestor of u (sym., v) in the same stratum. Each node x at the *top* of a stratum (such that $p(x)$ is in a higher stratum) maintains a set $S'(x)$ containing corresponding entries for edges incident on descendents in deeper strata. Thus, each edge occurs in only $O(\sqrt{D})$ sets. $C(T)$ is similarly stratified.

Every query on a set $S(x)$ in the above discussions is answered by uniting the results from the same queries on the new $S(x)$ and $S'(\mathit{sr}(x))$, where $\mathit{sr}(x)$ is the ancestor of x at the top of x's stratum.

Let the data structure underlying the $S(\cdot)$ sets (e.g., a CST or PST) use $\mathcal{S}(m)$ space, $\mathcal{Q}(m)$ query time, $\mathcal{R}(m)+k$ reporting time, and $\mathcal{U}(m)$ update time or, in the static case, $\mathcal{P}(m)$ preprocessing time. Let D be the depth of the tree being stratified (either T or $C(T)$).

Theorem 4. *A data structure using $O(\sqrt{D}\mathcal{S}(m))$ space can be built to support edgeQuery$(\cdot)$ in $O(\mathcal{Q}(m))$ time, edgeExpand$(\cdot,\cdot)$ in $O(k\mathcal{Q}(m))$ time, and edgeReport$(\cdot)$ in $O(\mathcal{R}(m)+k)$ time, where k is the number of edges reported. In the dynamic case, insertion and deletion of an edge into G take $O(\sqrt{D}\mathcal{U}(m))$ time; in the static case, the preprocessing time is $O(\sqrt{D}\mathcal{P}(m))$.*

We can also stratify recursively. Starting with the one-level stratification above, stratify the $\sqrt{D}$ stratum top nodes into $D^{1/4}$ strata of $D^{1/4}$ levels each. Similarly recurse on the nodes within each stratum. We can doubly recurse $\log\log D$ levels until there remain $O(1)$ strata containing $O(1)$ nodes each. Each edge is thus recorded in $O(\log D)$ $S(\cdot)$ and $S'(\cdot)$ sets.

Theorem 5. *A data structure using $O(\mathcal{S}(m)\log D)$ space can be built to support edgeQuery$(\cdot)$ in $O(\mathcal{Q}(m)\log D)$ time, edgeExpand$(\cdot,\cdot)$ in $O(k\mathcal{Q}(m)\log D)$ time, and edgeReport$(\cdot)$ in $O(\mathcal{R}(m)\log D + k)$ time, where k is the number of edges reported. In the dynamic case, insertion and deletion of an edge into G take $O(\mathcal{U}(m)\log D)$ time; in the static case, the preprocessing time is $O(\mathcal{P}(m)\log D)$.*

4.3 Higher Dimensions. The d-dimensional data structure on nodes of T_1 is a collection of sets $S_d(\cdot)$, such that $S_d(u_1)$ maintains the information as detailed above on hyperedges incident on descendents of $u_1 \in V(T_1)$. Consider such a hyperedge $(x_1, \ldots, x_d)$ $(x_1 \in \mathit{desc}(u_1))$. $S_d(u_1)$ is implemented recursively, as a collection of $S_{d-1}(\cdot)$ sets recording the projections $(x_2, \ldots, x_d)$ of the original hyperedges in $S_d(u_1)$. $S_2(\cdot)$ is the base case, equivalent to the $S(\cdot)$ sets above. There is a separate recursive collection of $S_{i-1}(\cdot)$ sets for each $S_i(\cdot)$ set; no space is allocated for empty sets.

This strategy allows for all except *edgeExpand*$(\cdot, 1)$ operations. If necessary, we maintain a second set of $S_d(\cdot)$ sets, designating a different tree to be T_1.

We stratify as in Section 4.2. Recall that in the one-level stratification, each original operation engendered two new operations (on the $S(\cdot)$ and $S'(\cdot)$ sets). Denote by $\mathcal{S}_d(m)$, $\mathcal{Q}_d(m)$, $\mathcal{R}_d(m) + k$, $\mathcal{U}_d(m)$, and $\mathcal{P}_d(m)$ the space and query, reporting, update, and preprocessing time bounds, rsp., for the d-dimensional tree-cross product operations. Let D be the depth of the tree (T or $C(T)$). Without stratification, we derive that $\mathcal{S}_d(m) = D\mathcal{S}_{d-1}(m)$, and $\mathcal{Q}_d(m) = \mathcal{Q}_{d-1}(m)$. With one-level stratification, we derive that $\mathcal{S}_d(m) = \sqrt{D}\mathcal{S}_{d-1}(m)$, and $\mathcal{Q}_d(m) = 2\mathcal{Q}_{d-1}(m)$. With recursive stratification, we derive that $\mathcal{S}_d(m) = \log D\mathcal{S}_{d-1}(m)$, and $\mathcal{Q}_d(m) = \log D\mathcal{Q}_{d-1}(m)$. With all methods, the derivation for $\mathcal{R}_d(m)$ follows that for $\mathcal{Q}_d(m)$, and those for $\mathcal{U}_d(m)$ and $\mathcal{P}_d(m)$ follow that for $\mathcal{S}_d(m)$.

Table 1 details some of the resulting bounds, using compressed trees (hence $D = O(\log n)$) and either Overmars' static three-sided range query structure [23] for the static case or Willard's PST [29] for the dynamic case.

These results strictly improve upon what we could derive using classical range searching on the original grid. Consider the two-dimensional case, for example. Overmars' static structure [23] would match only our non-stratified, static space and query time bounds, but his preprocessing time is significantly higher; to reduce the latter would degrade the query time by a $\sqrt{\log n}/\log\log n$ factor. Applying Edelsbrunner's technique [10] to Willard's PST [29] would match only our non-stratified, dynamic bound. Stratification improves all these bounds. We also provide a dynamic solution that achieves logarithmic query **and** update times in almost-linear space. No classical range searching result provides the same bounds. Chazelle [8] provides linear space bounds, but the query and update times are $O(\log^2 n)$, and reporting imposes a non-constant penalty on k.

Table 1. Deterministic, worst-case bounds for d-dimensional tree cross-product operations. $n = |V(G)|$, $m = |E(G)|$. For all methods, the *edgeReport*$(\cdot,\cdot)$ time is $k + f(n)$, and the *edgeExpand*$(\cdot,\cdot)$ time is $k \cdot f(n)$, where k is the number of edges reported, and $f(n)$ is the corresponding *edgeQuery*$(\cdot,\cdot)$ time.

Method	Space	*edgeQuery*$(\cdot,\cdot)$ time	
Static			Preproc. time
No stratification	$O(m \log^{d-1} n)$	$O(\log \log n)$	$O(m \log^{d-1} n)$
One-level strat.	$O(m \log^{\frac{d-1}{2}} n)$	$O(2^{d-1} \log \log n)$	$O(m \log^{\frac{d-1}{2}} n)$
Recursive strat.	$O(m(\log \log n)^{d-1})$	$O((\log \log n)^d)$	$O(m(\log \log n)^{d-1})$
Dynamic			Update Time
No stratification	$O(m \log^{d-1} n)$	$O(\log n / \log \log n)$	$O(\log^d n / \log \log n)$
One-level strat.	$O(m \log^{\frac{d-1}{2}} n)$	$O(2^{d-1} \log n / \log \log n)$	$O(\log^{\frac{d+1}{2}} n / \log \log n)$
Recursive strat.	$O(m(\log \log n)^{d-1})$	$O(\log n (\log \log n)^{d-2})$	$O(\log n (\log \log n)^{d-2})$

5 Applications

5.1 Hierarchical Graph Views. Given a rooted tree T and a graph G, such that the vertices of G correspond to the leaves of T, we say that $U \subseteq V(T)$ *partitions* G if the set $\{\mathit{leaves}(v) : v \in U\}$ partitions $V(G)$. A *view of* G is any $U \subseteq V(T)$ that partitions G. We extend the definitions from Section 2. For any $u, v \in V(T)$ such that neither u nor v is an ancestor of the other, define $E(u,v) = \{\{x,y\} \in E(G) : x \in \mathit{leaves}(u) \wedge y \in \mathit{leaves}(v)\}$. If each edge $e \in E(G)$ has a weight, $w(e) \in \mathcal{G}$, from an abelian group, $(\mathcal{G}, \odot)$, then the *weight* of (u,v) is $w(u,v) = \bigodot \{w(\{x,y\}) : \{x,y\} \in E(u,v)\}$. For any view U, we define G/U to be the *induced graph* (U, E_U), where $E_U = \{(u,v) \in U \times U : |E(u,v)| > 0\}$.

The *hierarchical graph-view maintenance problem* is to maintain G/U under the following operations on U: *expand*(U,x), where $x \in U$, yields view $U \setminus \{x\} \cup \mathit{children}(x)$; *contract*$(U,x)$, where $\mathit{children}(x) \subseteq U$, yields view $U \setminus \mathit{children}(x) \cup \{x\}$. The problem is motivated by graph visualization applications [7, 9].

Let $n = |V(G)|$, $m = |E(G)|$, $p = \min\{\mathit{depth}(T), \log n\}$, and assume without loss of generality that T contains no unary vertices. (Hence $|V(T)| = O(n)$.) Buchsbaum and Westbrook [7] show how, with $O(mp)$ space and $O(mp^2)$ expected preprocessing time, to perform *expand*$(\cdot,\cdot)$ and *contract*$(\cdot,\cdot)$ operations in optimal time: linear in the number of changes to $E(G/U)$. There is an additional $\log n$ factor in the *expand*$(\cdot,\cdot)$ time for weighted graphs. To accommodate updates to G, the space bound becomes $O(mp^2)$, the edge insertion and deletion times are expected $O(p^2 \log n)$; *expand*$(\cdot,\cdot)$ and *contract*$(\cdot,\cdot)$ remain optimal.

By applying tree cross products, we improve the space, update and preprocessing times for unweighted graph-view maintenance. The cost is an increase in *expand*$(\cdot,\cdot)$ time; *contract*$(\cdot,\cdot)$ remains optimal. All of our bounds are deterministic, worst-case; the prior update and preprocessing times [7] are expected.

Set $T_1 = T_2 = T$. Edge $(u,v) \in E(G)$ (ordered by postorder on T) becomes an edge from u in T_1 to v in T_2. An *expand*(U,v) engenders an *edgeExpand*$((u,v),v)$ operation for each $(u,v) \in E(G/U)$ (and symmetrically for $(v,w) \in E(G/U)$).

Induced edges between children of v are found using nearest common ancestors [7]. To implement *contract*(U, v), add an edge to v from each non-child of v adjacent to a child of v in G/U, and remove edges incident on children of v.

Define $\mathrm{Opt}(U, v)$ to be the number of nodes adjacent to children of v in G/U. Denote by U the view before an *expand*$(U, \cdot)$ or *contract*$(U, \cdot)$ and U' the result.

Theorem 6. *On an unweighted graph, with $O(m \log\log n)$ space, we can perform edge insertion and deletion in $O(\log n)$ time, expand(U, v) in $O(\mathrm{Opt}(U', v) \cdot \log n)$ time, and contract(U, v) in $O(\mathrm{Opt}(U, v))$ time. In the static case, with $O(m \log\log n)$ space and preprocessing time, we can perform expand(U, v) in $O(\mathrm{Opt}(U', v) \cdot (\log\log n)^2)$ time and contract(U, v) in $O(\mathrm{Opt}(U, v))$ time.*

Theorem 6 follows from Theorem 5, assuming recursive stratification. One-level stratification improves the *expand*(U, v) times by $\log\log n$ factors but degrades the space, update and preprocessing times by $\sqrt{\log n}/\log\log n$ factors.

5.2 String Matching. Given text $T = x_1 \cdots x_n$ of length n, denote by $T[i, j]$ the substring $x_i \cdots x_j$. The *text indexing with one error problem* is to preprocess T so that, given length-m pattern P, we can compute all locations i in T such that P matches $T[i, i+m-1]$ with exactly one error. Below we assume Hamming distance, i.e., the number of symbols replaced, but the method extends to Levenshtein (edit) distance [19].

This on-line problem differs from *approximate string matching* [14, 26], in which both T and P are given off-line. Exact text indexing (finding occurrences of P with no errors), can be solved with $O(n)$ preprocessing time and space, $O(m+k)$ query time, where k is the number of occurrences [21, 28], and $O(\log^3 n + s)$ time to insert or delete a length-s substring in T [26].

The work of Ferragina, Muthukrishnan, and de Berg [11] extends to solve text indexing with one error. Given $O(n^{1+\varepsilon})$ preprocessing time and space, queries take $O(m \log\log n + k)$ time. Using the same approach, Amir et al. [2] give a solution with $O(n \log n)$ preprocessing time and space but $O(m\sqrt{\log n}+k)$ query time. Both assume no exact matches occur. We improve these results, achieving $O(m \log\log n + k)$ query time and $O(n\sqrt{\log n})$ space and preprocessing time. Amir et al. [2] extend their solution to the general case with $O(n \log^2 n)$ space and preprocessing time and $O(\log n \log\log n + k)$ query time. We similarly extend our solution, achieving $\log n$ factor improvements in all bounds.

Observe [2, 11] that an occurrence of P in T at location i with one error at location $i+j$ implies that $T[i, i+j-1]$ matches $P[1, j]$ and $T[i+j+1, i+m-1]$ matches $P[j+2, m]$. To exploit this, Amir et al. [2] first build suffix trees S_T for T and S_{T^R} for T^R, the reverse string of T, using Weiner's method [28]. Label each leaf in S_T by the starting location of its suffix in T; label each leaf in S_{T^R} by $n - i + 3$, where i is the starting location of the corresponding suffix in T^R. Querying for P is done as follows.

For $j = 1, \ldots, m$ do

1. Find node u, the location of $P[j+1, m]$ in S_T, if such a node exists.
2. Find node v, the location of $P[1, j-1]^R$ in S_{T^R}, if such a node exists.
3. If u and v exist, report the intersection of the labels of *leaves*(u) and *leaves*(v).

Steps (1) and (2) can be performed in $O(m)$ time over the progression of the algorithm [2], by implicitly and incrementally continuing the Weiner construction [28] on the suffix trees. By adding edges connecting pairs of identically labeled leaves, Step (3) becomes an *edgeReport*(u, v) operation.

Theorem 7. *Given $O(n\sqrt{\log n})$ preprocessing time and space, we can preprocess a string T of length n, so that, for any string P of length m given on-line, if no exact matches of P occur in T, we can report all occurrences of P in T with one error in $O(m \log\log n + k)$ time, where k is the number of occurrences.*

Each exact match would be reported $|P|$ times. To obviate this problem, we add a third dimension as do Amir et al. [2]. Tree T_3 contains a root and s leaves, each corresponding to one of the $s \leq n$ alphabet symbols. Each edge connecting leaves in S_T and S_{T^R} corresponds to some mismatch position i in T. We extend the (hyper)edge to include leaf $T[i, i]$ in T_3. We extend the *edgeReport*$(\boldsymbol{u})$ semantics to allow the stipulation that any dimension j report elements that are *not* descendents of u_j. (This simply changes the parameters of the queries performed on the $S(\cdot)$ sets.) Step (3) becomes an *edgeReport*$(u, v, T[i, i])$ operation.

Theorem 8. *Given $O(n \log n)$ preprocessing time and space, we can preprocess a string T of length n, so that, for any string P of length m given on-line, we can report all k occurrences of P in T with one error in $O(m \log\log n + k)$ time.*

Grossi and Vitter [15] use a similar strategy to report contiguous ranges in *compressed suffix arrays*, which use only $O(n)$ **bits** to implement all suffix-array functionality on a length-n binary string T. They use two-dimensional, grid range searches that can be equivalently realized by node-intersection queries on suffix trees for T and T^R. As above, tree cross products improve their bounds on the additional suffix-array reporting time, from $O(\log^2 n \log\log n + k)$ to $O(\log n (\log\log n)^3 + k)$, where k is the output size.

5.3 Hammocks. Let $G = (V, E)$ be a directed graph with a designated source node, s, and sink node, t. A node u *dominates* a node v if every path from s to v goes through u. A node v *post-dominates* a node u if every path from u to t goes through v. The *hammock* between two nodes u and v is the set of nodes dominated by u and post-dominated by v. (This modifies the definition due to Kas'janov [18].)

Johnson, Pearson, and Pingali [17] define a canonical, nested hammock structure, which is useful in compiler control-flow analysis, and devise an $O(m)$-time algorithm to discover it. ($n = |V|$, and $m = |E|$.) No previous result, however, allows efficient, on-line queries of the form: return the hammock between two given nodes. Such queries are useful in software system analysis, to detect collections of systems with designated information choke points, e.g., to assess the impact of retiring legacy systems [5].

We can solve such queries as follows. Let T_1 be the dominator tree [1, 6] of G, and let T_2 be the dominator tree of the reverse graph of G. The hammock between two nodes u and v in G is the intersection of the set of descendents of u in T_1

with the set of descendents of v in T_2. By adding edges connecting corresponding nodes in T_1 and T_2, this intersection is computed by an *edgeReport*(u, v) query.

Theorem 9. *With $O(n\sqrt{\log n})$ (rsp. $O(n \log\log n)$) space and preprocessing time, we can compute the hammock between two given nodes in $O(\log\log n + k)$ (rsp., $O((\log\log n)^2 + k)$) time, where k is the size of the hammock.*

6 Conclusion

Many applications impose balance or low-depth constraints on T, which obviate the sophisticated space-reduction techniques and allow the $S(\cdot)$ sets to be implemented by simple binary search trees, making our tree cross-product framework very practical. Low-degree constraints on T might lead to other simplifications.

How to implement *edgeWeight*$(\cdot, \cdot)$ operations efficiently remains open. It also remains open to unify our graph-view bounds with those of Buchsbaum and Westbrook [7], i.e., to eliminate the penalty that we incur on expand times.

Finally, allowing updates to T remains open.

Acknowledgements. We thank Raffaele Giancarlo and S. Muthukrishnan for helpful discussions and Roberto Grossi for tutelage on compressed suffix arrays.

References

1. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM J. Comp.*, 28(6):2117–32, 1999.
2. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. In *Proc. 6th WADS*, volume 1663 of *LNCS*, pages 181–92. Springer-Verlag, 1999.
3. R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. 7th CPM*, volume 1075 of *LNCS*, pages 1–23. Springer-Verlag, 1996.
4. R. A. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. 5th WADS*, volume 1272 of *LNCS*, pages 174–84. Springer-Verlag, 1997.
5. A. L. Buchsbaum, Y. Chen, H. Huang, E. Koutsofios, J. Mocinego, A. Rogers, M. Jenkowsky, and S. Mancoridis. Enterprise navigator: A system for visualizing and analyzing software infrastructures. Technical Report 99.16.1, AT&T Labs–Research, 1999. Submitted for publication.
6. A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM TOPLAS*, 20(6):1265–96, 1998. *Corrigendum*, 27(3):383–7, 2005.
7. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proc. 11th ACM-SIAM SODA*, pages 566–75, 2000.
8. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comp.*, 17(3):427–62, 1988.
9. C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees and their use for drawing very large graphs. In *Proc. GD '98*, volume 1547 of *LNCS*, pages 111–24. Springer-Verlag, 1998.

10. H. Edelsbrunner. A note on dynamic range searching. *Bull. EATCS*, 15:34–40, 1981.
11. P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *Proc. 31st ACM STOC*, pages 483–91, 1999.
12. O. Fries, K. Mehlhorn, S. Näher, and A. Tsakalidis. A $\log \log n$ data structure for three-sided range queries. *IPL*, 25:269–73, 1987.
13. H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. 1st ACM-SIAM SODA*, pages 434–43, 1990.
14. Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *J. Complexity*, 4:33–78, 1988.
15. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM STOC*, pages 397–406, 2000.
16. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comp.*, 13(2):338–55, 1984.
17. R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proc. ACM SIGPLAN PLDI '94*, pages 171–85, 1994.
18. V. N. Kas'janov. Distinguishing hammocks in a directed graph. *Sov. Math. Dokl.*, 16(2):448–50, 1975.
19. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dok.*, 10:707–10, 1966.
20. U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *IPL*, 50(4):191–7, 1994.
21. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–72, 1976.
22. R. Muth and U. Manber. Approximate multiple string search. In *Proc. 7th CPM*, volume 1075 of *LNCS*, pages 75–86. Springer-Verlag, 1996.
23. M. H. Overmars. Efficient data structures for range searching on a grid. *J. Alg.*, 9:254–75, 1988.
24. P. A. Pevzner and M. S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 12(1/2):135–54, 1995.
25. F. P. Preparata, J. S. Vitter, and M. Yvinec. Output-sensitive generation of the perspective view of isothetic parallelepipeds. *Algorithmica*, 8(4):257–83, 1992.
26. S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proc. 37th IEEE FOCS*, pages 320–8, 1996.
27. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
28. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Symp. on Switch. and Auto. Thy.*, pages 1–11, 1973.
29. D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comp.*, 29(3):1030–49, 2000.