An Experimental Analysis of Change Propagation in Dynamic Trees *

Umut A. Acar † Guy E. Blelloch ‡ Jorge L. Vittes §

Abstract

Change propagation is a technique for automatically adjusting the output of an algorithm to changes in the input. The idea behind change propagation is to track the dependences between data and function calls, so that, when the input changes, functions affected by that change can be re-executed to update the computation and the output. Change propagation makes it possible for a compiler to dynamize static algorithms. The practical effectiveness of change propagation, however, is not known. In particular, the cost of dependence tracking and change propagation may seem significant.

The contributions of the paper are twofold. First, we present some experimental evidence that change propagation performs well when compared to direct implementations of dynamic algorithms. We implement change propagation on tree-contraction as a solution to the dynamic trees problem and present an experimental evaluation of the approach. As a second contribution, we present a library for dynamic-trees that support a general interface and present an experimental evaluation by considering a broad set of applications. The dynamic-trees library relies on change propagation to handle edge insertions/deletions. The applications that we consider include path queries, subtree queries, least-common-ancestor queries, maintenance of centers and medians of trees, nearest-marked-vertex queries, semi-dynamic minimum spanning trees, and the max-flow algorithm of Sleator and Tarjan.

1 Introduction

Change propagation is a technique for automatically updating the output of an algorithm or program according to a change in the input. A recent paper [1] introduces change propagation and studies its semantics and correctness. A subsequent paper [2] presents analysis techniques for analyzing the performance change propagation and applies change propagation to the dynamic-trees problem [17].

Change propagation relies on keeping a graph of dependences between code and data. Whenever a memory location is read, a dependence is created between that location and the function that reads the location. Once a computation is performed, any value stored in memory (in particular, the inputs) can be changed and the computation can be updated by propagating the changes through memory by a change-propagation algorithm.

The *change-propagation algorithm* maintains a set of functions affected by the changes and re-executes them in sequential-execution order. When a re-executed function changes a memory location by writing to it, all functions that read the changed location become affected. Since re-execution can create new reads and obliterate previously performed reads due to conditionals, the dependences themselves change dynamically. Since dependence tracking and change propagation are automatic, a compiler can be used to dynamize static algorithms. Change propagation can therefore dramatically simplify the design and implementation of algorithms and data structures for handling changing inputs such as dynamic and kinetic data structures.

In previous work [2] we showed that applying change propagation on static tree-contraction algorithm of Miller and Reif [14] yields a solution to the dynamic trees-problem of Sleator and Tarjan [17]. Dynamic-trees data structures are fundamental structures that arise as a substep in many important algorithms such as dynamic graph algorithms [12, 6, 11, 8], max-flow algorithms [17, 18, 9], computational geometry algorithms [10]. Several dynamic-tree data structures have been proposed including Link-Cut Trees [17, 18], Euler-Tour Trees [11, 20], Topology Trees [8], and Top Trees [3, 4, 19].

Our approach is to map a forest of possibly unbalanced trees, called *primitive trees*, to a set of balanced trees, called *RC-Trees* (Rake-and-Compress Trees) by applying tree-contraction technique of Miller and Reif [14] on each primitive tree, and use change propagation to update the RC-Trees when edges are

*This work was supported in part by the National Science Foundation through the Aladdin Center (www.aladdin.cs.cmu.edu) under grants CCR-0085982 and CCR-0122581. The third author was also supported by an NSF REU through the Aladdin Center.

†Toyota Technological Institute, Chicago, IL 60637. umut@tti-c.org.

‡Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. blelloch@cs.cmu.edu

§Computer Science Department, Stanford University, Stanford, CA 94305. jvittes@stanford.edu.

inserted/deleted dynamically [2] into/from the primitive trees. To process applications-specific queries we annotate RC-Trees with application-specific data and use standard techniques to compute queries. To process application-specific data changes, we recompute the annotations of the RC-Trees that are affected by the change.

Since we rely on tree-contraction to map unbalanced trees to balanced tree, the primitive trees (the trees being mapped) must be bounded degree. Therefore, all our experiments involve bounded-degree trees. Since any tree can be represented as a bounded-degree tree, this restriction causes no loss of generality. In particular, RC-Trees can be applied to arbitrary trees by mapping an arbitrary tree T to a bounded-degree tree T' by splitting high-degree nodes into bounded-degree nodes [7], and by building an RC-Tree based on the bounded-degree tree T'. To support dynamic-tree operations, the mapping between T and T' must be maintained dynamically as the degree of the vertices in T change due to insertions/deletions by joining and splitting the vertices of T'. In certain applications, it is also possible to preprocess the input to ensure that bounded-degree dynamic-tree data structures can be used directly (without having to split and merge vertices dynamically). For example, the input network for a max-flow algorithm can be preprocessed by replacing high-degree vertices with a network of bounded-degree vertices that are connected via infinite capacity edges.

The advantages of change propagation include,

- composibility: the ability to input the result of one dynamic algorithm to another while ensuring that all changes are propagated,
- the ability to handle batch changes, and
- ease of programming.

Since dependence tracking and change propagation can be completely automated [2, 1], the static-to-dynamic transformation can be done by a compiler. In this paper, we perform the transformation semi-automatically as described in Section 5.

Contributions of this Paper. The objectives of this paper are to assess practical effectiveness of change propagation, and to provide an implementation and evaluation of a dynamic-trees data structure that supports a broad range of applications. The code for the implementation is publicly available through the Aladdin Center web site at `www.aladdin.cs.cmu.edu`.

Beyond the focus on change propagation, our implementation and experiments have the following properties:

1. The original data structure [2] uses $O(n \log n)$ space and is strongly history independent. We present an $O(n)$-space implementation that is history independent (oblivious) as defined by Micciancio [13]. The implementation can be made weakly history independent according to the definition of history independence by Naor and Teague [15] within the same time bounds by using their memory allocator.
2. The data structure separates *structural* operations (links and cuts) from *application-specific* queries and data changes: the structure (*i.e.*, shape, layout) of the data structure depends only on the structure of the underlying tree and is changed only by link and cut operations, applications-specific queries and data changes merely traverse the RC-Trees and change the annotations (tags).

   Previous data structures such as Link-Cut Trees [17, 18] and Top-Trees [4, 3] do allow the structure of the data structure to be modified by application queries via the *evert* and the *expose* operations. For example, in Link-Cut trees, one node of the underlying tree is designated as root, and all path queries must be with respect to this root. When performing a path query that does not involve the root, one end point of the path query must be made root by using the evert operation. In top trees, the user operates on the data structure using a set of operations provided by the interface and is expected to access only the root node of the data structure. The user ensures that the root note contains the desired information by using the *expose* operation. The expose operation often restructures the data structure. For example, in top trees, a path query that is different than the previous path query requires an expose operation that restructures the data structure.
3. The data structure directly supports batch changes.
4. We present an experimental evaluation by considering a large number of applications including
   - path queries as supported by Link-Cut Trees [17, 18],
   - subtree queries as supported by Euler Tour Trees [11, 20],
   - non-local search: centers, medians, and least-common-ancestors as supported by Top Trees [4, 3],
   - finding the distance to a set of marked vertices from any source,
   - semi-dynamic (incremental) minimum spanning trees, and

- max-flow algorithm of Sleator and Tarjan [17].

Our experiments confirm the theoretical time bounds for change propagation and show that the approach can yield efficient dynamic algorithms even when compared to direct implementations. The experiments show that the full separation of application-specific operations from structural operations significantly speed up application-specific operations such as queries and data changes. When compared to a direct implementation of Link-Cut Trees [17, 18], path queries are up to a factor of three faster, whereas structural operations are up to a factor of five slower. This trade-off can make one data structure preferable to the other depending on the application. For example, an incremental minimum-spanning tree algorithm can perform many more queries than insertions and deletions, whereas a max-flow algorithm is likely to perform more structural operations.

A key property of change propagation is that it directly supports batch processing of changes without requiring any change to the implementation. In the context of dynamic-trees, our experiments show a large performance gap (asymptotically a logarithmic factor) between batch and one-by-one processing of changes as would be required in previously proposed data structures. *Batch changes* arise in the following two settings:

1. *multiple, simultaneous changes to the input:* For example, the user of a dynamic minimum-spanning tree algorithm may choose to delete many edges at once. In this case, it would be more efficient to process all changes by a single pass on the data structure than to break them up into single changes.

2. *propagation of a single change to multiple changes:* For example, the dynamic connectivity and the Minimum-Spanning Tree algorithms of Holm *et al* [12], maintain a hierarchy of dynamic trees. When a single edge is deleted from the input graph, a number of edges in one level of the hierarchy can be inserted into the next level. Since the dynamic-trees data structure for the next level is never queried until all insertions are performed, the insertions can be processed as a batch. In general, supporting batch changes is essential if dynamic data structures are to be composed, because a single change can propagate to multiple changes.

2 Rake-and-Compress Trees

This section describes a solution to the dynamic-trees problem based on change propagation on a static tree-contraction algorithm. The section summarizes the previous work [2] and gives more detail on the processing of application-specific queries.

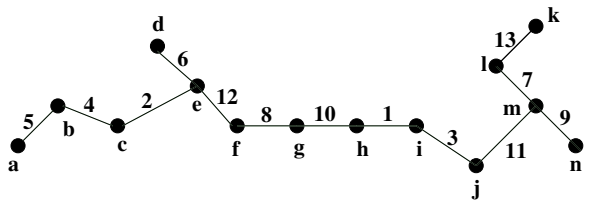


Figure 1: A weighted tree.

We use Miller and Reif's tree-contraction algorithm [14] to map arbitrary, bounded-degree trees to a balanced trees. We call the trees being mapped, the *primitive trees*, and the balanced trees that they are mapped to, the *RC-Trees* (Rake-and-Compress Trees). RC-Trees represents a recursive clustering of primitive trees. Every node of an RC-Tree represents a cluster (subtree) of the corresponding primitive tree.

To apply RC-Trees to a particular problem, we annotate RC-Trees with information specific to that problem, and use tree traversals on the RC-Trees to compute properties of the primitive trees. Tree traversals on RC-Trees alone suffice to answer dynamic queries on static primitive trees. To handle dynamic changes to primitive trees, *i.e.*, edge insertions and deletions, we use change propagation. Given a set of changes to a primitive tree, the change-propagation algorithm updates the RC-Tree for that primitive tree by rebuilding the parts of the RC-Tree affected by the change.

2.1 Tree Contraction and RC-Trees. Given some tree T, the tree-contraction algorithm of Miller and Reif [14] contracts T to a single vertex by applying rake and compress operations over a number rounds. *Rake* operations delete all leaves in the tree and *compress* operations delete an independent set of vertices that lie on *chains*, *i.e.*, paths consisting of degree two vertices. When the tree is contracted into a single vertex, a special *finalize* operation is performed to finish tree contraction. When using the random-mate technique, tree contraction takes expected linear time, and requires logarithmic number of rounds (in the size of the input) in expectation [14].

The original tree-contraction algorithm of Miller and Reif was described for directed trees. In this paper, we work with undirected trees and use the generalization tree contraction for undirected trees [2]. For applications, we assume that the edges and vertices of trees can be assigned *weights* (for edges) and *labels* (for vertices) respectively. Weights and labels can be of any type.

As an example consider the weighted tree shown in

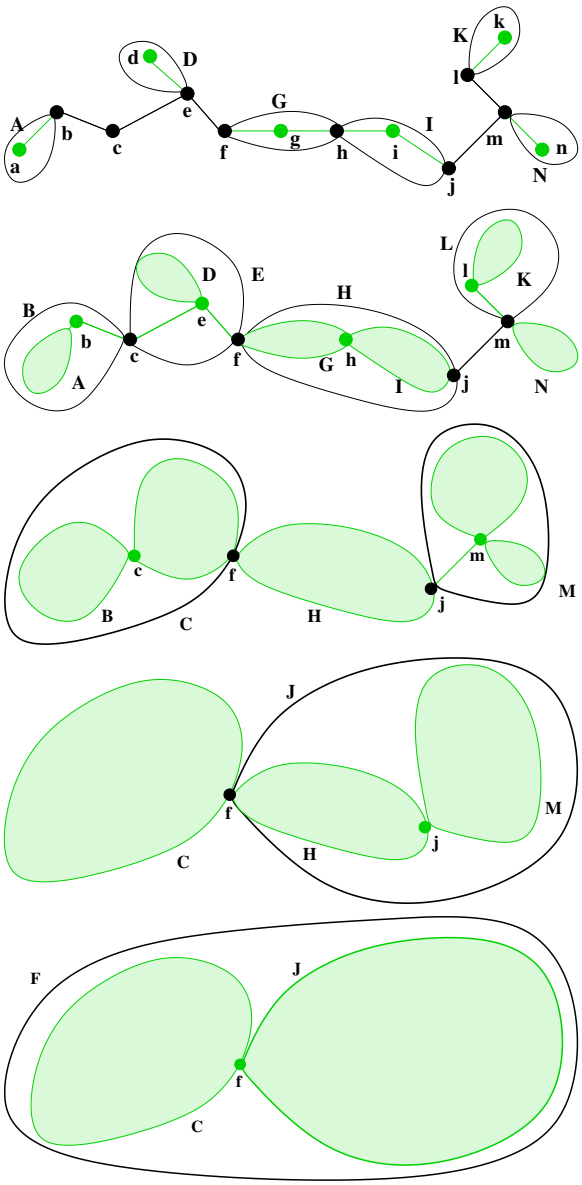


Figure 2: An example tree-contraction.

Figure 1. Figure 2 shows the complete contraction of the tree (the rounds increase from top to bottom). Since tree contraction does not depend on the weights they are omitted from Figure 2. Each round of a contraction deletes a set of vertices by using the rake and compress operations.

— A *rake* operation deletes a leaf and the edge adjacent to it, and stores the contribution of the deleted vertex and the edge in its neighbor. The *contribution* of a raked vertex is computed by calling the `rake_data` operation that is specified by the application. The `rake_data` operation computes the contribution from 1) the label of the deleted vertex, 2) contributions stored at that vertex, and 3) the weight of the deleted edge.

— A *compress* operation removes a degree two vertex, say v, and the edges adjacent to it, (u, v) and (v, w), and inserts an edge (u, w), and stores the contribution of v, (u, v), and (v, w) on u and w. The *contribution* of a compressed vertex is computed by calling the `compress_data` operation that is specified by the application. The `compress_data` operation computes the contributions from 1) the label of v, 2) the contributions stored at v, and 3) the weights of (u, v) and (v, w).

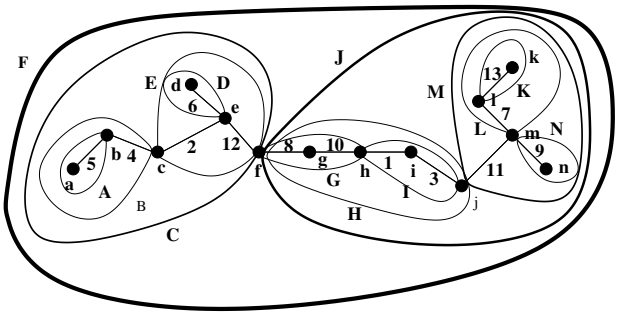


Figure 3: A clustering.

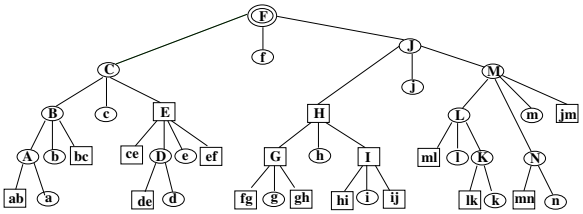


Figure 4: An RC-Tree.

For example, in Figure 2, the first round rakes the vertices a, d, n, and k and compresses g, and i. At the end of the contraction, when the tree is reduced to a single vertex, a special *finalize* operation is performed to compute a value from the contributions stored at that vertex by calling the `finalize_data` operation that is specified by the application.

Tree contraction can be viewed as recursively clustering a tree into a single cluster. Initially the vertices and the edges form the *base clusters*. The rake, compress, and finalize operations form larger clusters by a number of smaller clusters and a base cluster consisting of a single vertex. In Figure 2, all clusters (except for the base clusters) are shown with petals. Each cluster is labeled with the capitalized label of the vertex

joining into that cluster. For example, raking vertex a creates the cluster A consisting of a and b, and the edge (a, b); compressing the vertex g creates the cluster G, consisting of the vertex g and the edges (f, g) and (g, h). In the second round, raking the vertex b creates the cluster B that contains the cluster A and the edge (b, c). In the last round, finalizing f creates the cluster F that contains the clusters C and J. Figure 3 shows the *clustering* consisting of all the clusters created by the contraction shown in Figure 2.

We define a *cluster* as a subtree of the primitive tree induced by a set of vertices. For a cluster C, we say that vertex v of C is a *boundary vertex* if v is adjacent to a vertex that does not belong to C. The *boundary* of a cluster consists of the set of boundary vertices of that cluster. The *degree* of a cluster is the number vertices in its boundary. For example, in Figure 3, the cluster A has the boundary $\{b\}$, and therefore has degree one; the boundary of the cluster G is $\{f, g\}$ and therefore G has degree two. In tree contraction, all clusters except for the final cluster has degree one or degree two. We will therefore distinguish between *unary, binary, and final* clusters. It is a property of the tree contraction that

1. the rake operations yield unary clusters,
2. the compress operations yield binary clusters, and
3. the finalize operation yields the final cluster which has degree zero.

The contraction of a primitive tree can be represented by a tree, called *RC-Tree*, consisting of clusters. Figure 4 shows the RC-Tree for the example contraction shown in Figures 2 and 3. When drawing RC-Trees, we draw the unary clusters with circles, the binary clusters with squares, and the final cluster with two concentric circles. Since tree contraction takes expected logarithmic number of rounds, the RC-Tree of a primitive tree is probabilistically balanced. It is in this sense, that tree contraction maps unbalanced trees to balanced trees.

Throughout this paper, we use the term "node" with RC-Trees and the term "vertex" with the underlying primitive trees. We do not distinguish between a node and the corresponding cluster when the context makes it clear.

2.2 Static Trees and Dynamic Queries. RC-Trees can be used to answer dynamic queries on static trees. The idea is to annotate the RC-Trees with application-specific information and use tree traversals on the annotated trees to answer application-specific queries. To enable annotations, we require that the applications provide `rake_data`, `compress_data`, and `finalize_data` functions that describe how data (edge weights and/or vertex labels) is combined during rake, compress, and finalize operations respectively. Using these operations, the tree-contraction algorithm tags each cluster with the value computed by the `rake_data`, `compress_data`, and `finalize_data` operation computed during the operation that forms that cluster.

Once an RC-tree is annotated, it can be used to compute various properties of the corresponding primitive tree by using standard tree traversals. For all applications considered in this paper, a traversal involves a constant number of paths between the leaves and the root of an RC-Tree. Depending on the query, these paths can be traversed top down, or bottom-up, or both ways. Since RC-Trees are balanced with high probability, all such traversals require logarithmic time in expectation.

Section 3 describes some applications demonstrating how RC-Trees can be used to answer various queries such path queries, subtree queries, non-local search queries in expected logarithmic time.

2.3 Dynamic Trees and Dynamic Queries. To support dynamic trees, *i.e.*, edge insertion/deletions (link and cuts), we use change propagation. Change propagation effectively rebuilds the tree-contraction in expected logarithmic time by only rebuilding the clusters affected by the change [2] and computing their annotations.

To support *data changes*, we update the annotations in the tree by starting at the leaf of the RC-Tree corresponding to the changed vertex or edge and propagating this change up the RC-Trees. Since RC-Trees are balanced with high-probability this requires expected logarithmic time.

Theorem 2.1 (Dynamic Trees) *For a bounded-degree forest F of n vertices, an RC-Forest F_{RC} of F can be constructed and annotated in expected $O(n)$ time using tree-contraction and updated in expected $O(\log n)$ time under edge insertions/deletions, and application specific data changes. Each tree in the RC-Forest F_{RC} has height expected $O(\log n)$.*

3 Applications

This section describes how to implement a number of applications using RC-Trees. Experimental results with these application are given in Section 5. Implementing an application using RC-Trees involves

1. providing the `rake_data`, `compress_data`, and `finalize_data` operations that specify how data is combined during rake, compress, and finalize operations respectively, and

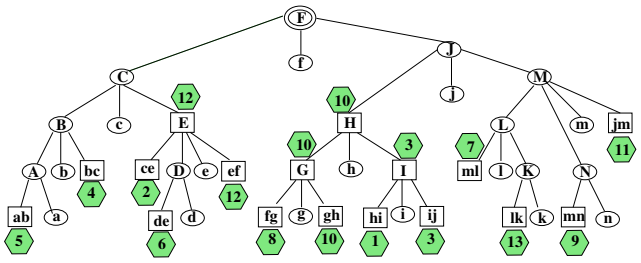


Figure 5: An RC-Tree with tags.

2. implementing the queries on top of RC-Trees using standard tree traversals.

This section considers the following applications: path queries, subtree queries, diameter queries, center/median queries, least-common-ancestor queries, and nearest-marked-vertex queries. Of these queries, path queries, subtree queries, diameter queries, and nearest-marked-vertex queries all require simple tree traversal techniques that traverse a fixed number of bottom-up paths in the RC-Tree. Other queries, including centers/medians and least-common ancestor queries, require traversing a fixed number of bottom-up and top-down paths. Queries that require top-down traversal are sometimes called non-local search queries [4], because they cannot be computed by only using local information pertaining to each cluster.

Throughout this paper, we work with undirected trees. Applications that rely on directed trees specify an arbitrary root vertex with each query. For the discussion, we define the *cluster path* of a binary cluster as the path between its two boundary vertices.

3.1 Path Queries. A path query [17, 18] asks for the heaviest edge on a path. Consider, as an example, the tree shown in Figure 1. The answer for the path query with vertices c and m is the edge (e, f) with weight 12. In general path queries can be defined on arbitrary associative operations on weights. The techniques described here apply in this general setting.

For this description, we assume that a path query asks for the weight of the heaviest edge on a given path—the heaviest edge itself can be computed by keeping track of edges in addition to weights. To answer path queries, we annotate each binary cluster with the weight of the heaviest edge on its cluster path. All other clusters (unary and final) will have no tags. These annotations require that the application programmer specify the functions `rake_data`, and `finalize_data` as "no-ops", and `compress_data` as the maximum operation on edge weights.

Figure 5 shows the RC-Tree for the contraction shown in Figure 3 where each binary cluster is annotated with the weight of the heaviest edge on its cluster path. For example, the cluster e is tagged with 12, because this is the largest weight on its cluster path consisting of the edges (c, e) and (e, f). The tag of each binary cluster is written in a hexagon. The RC-Tree is an annotated version of the tree in Figure 4.

By using the annotations, the maximum edge weight on the path between u and v can be found by simultaneously walking up the tree from u and v until they meet. For each cluster from u, we compute the maximum weight between u and the boundary vertices of that cluster. Similarly, for each cluster on the path from v, we compute the maximum weight between v and the boundary vertices of that cluster. Consider the cluster C that two paths meet and let C_u be the child of C on the path from u, and let C_v be the child of C on the path from v. The answer to the query is the maximum of the values computed for the common boundary vertex of C_u and C_v.

As an example, consider the primitive tree shown in Figure 1, its clustering Figure 3, the RC-Tree shown in Figure 5. Consider the path query with c and m. Consider the path from c to the root. The cluster C will be tagged with 12 because this is the heaviest weight between c and f (the boundary of C), and the cluster F will be tagged with 12. Consider now the path from m. The cluster M will be tagged with 11 and the cluster J will be tagged with 11. Since f is the first common boundary between the two paths explored, the result is 12, the maximum of the values associated with f, *i.e.*, 11 and 12.

3.2 Subtree Queries. A subtree query asks for the heaviest edge in a subtree. The subtree specified by a tree root r and a root v of the subtree. Consider, as an example, the tree shown in Figure 1. The answer for the path query with tree root c and subtree root m is the edge (l, k) with weight 13. In general subtree queries can be defined on arbitrary associative operations on weights. The technique described here applies in this general setting.

To answer subtree queries, we annotate each cluster with the weight of the heaviest edge in that cluster. These annotations require that the application programmer specifies the functions `rake_data`, `compress_data`, and `finalize_data` as the maximum operation.

By using the annotations, the maximum edge weight in a subtree specified by a tree root r and a subtree root v can be found by simultaneously walking up the tree from r and v until the paths meet. For the clusters on path from r, we compute no information.

For each cluster on the path from v, we compute the heaviest edge in the subtree rooted at v with respect to each boundary vertex as the tree root. Consider the RC-Tree node C that two paths meet and let C_r be the child of C on the path from r, and let C_v be the child of C on the path from v. The answer to the query is the maximum of the tag for C_r and the value computed for C_v for the common boundary vertex of C_r and C_v.

This technique for finding the heaviest edge in a subtree can be used to compute other kinds of subtree queries, such as queries that can be computed by Euler-Tour Trees [20, 11], by replacing the maximum operator with the desired associative operator.

3.3 Diameter. The diameter of a tree is the length of the longest path in the tree. For example, the longest path in the tree shown in Figure 1 is the path between a and k; therefore the diameter is 80. To compute the diameter of a tree, we annotate

— each unary cluster with its diameter, and the length of the longest path originating at its boundary vertex,

— each binary cluster with the length of its cluster path, its diameter, and length of the longest path originating at each boundary vertex, and

— the final cluster with its diameter.

It is relatively straightforward to specify the `rake_data`, `compress_data`, and `finalize_data` operations to ensure that the clusters are annotated appropriately.

Since the root of the RC-Tree is tagged with the diameter of the tree, computing the diameter requires no further traversal techniques. Note that, since change-propagation updates the annotations of the RC-Trees, the diameter will be updated appropriately when edge insertions/deletions take place.

3.4 Distance to the Nearest Marked Vertex. This type of query asks for the distance from a given query vertex to a set of predetermined marked vertices and has applications to metric optimization problems [4]. As an example, consider the tree shown in Figure 1 and suppose that the vertices a and n are marked. The answer to the nearest marked vertex query for vertex d is 17, because the length of the path between d and a is 17, whereas the length of the path between d and a is 60.

For this application, we annotate 1) each binary cluster with the length of its cluster path, and 2) each cluster with the distance between each boundary vertex and the nearest marked vertex in that cluster. The annotations are easily computed by specifying `rake_data`, `compress_data`, and `finalize_data` operations based on the minimum and addition operations on weights.

Given an annotated RC-Tree, we compute the distance from a query vertex v to the nearest marked vertex by walking up the RC-Tree from v as we compute the following values for each cluster on the path: 1) the minimum distance between v and the nearest marked vertex in that cluster, 2) the minimum distance between v and each boundary vertex. The answer to the query will be computed when the traversal reaches the root of the RC-Tree.

3.5 Centers and Medians. Centers and medians are non-local-search queries [4] and therefore require both a bottom-up and a top-down traversal of RC-Trees. They are quite similar, and therefore, we describe center queries only here.

For a tree T with non-negative edge weights, a center is defined as a vertex v that minimizes the maximum distance to other vertices in the tree. More precisely, let $d_T(v)$ be the maximum distance from vertex v to any other vertex in the tree T. A center of T is a vertex c such that $d_T(c) \leq d_T(v)$ for any vertex v.

To find the center of a tree, we annotate 1) each cluster C with $d_C(v)$ for each boundary vertex v of C, and 2) each binary cluster with the length of its cluster path. The annotations are easily computed by specifying `rake_data`, `compress_data`, and `finalize_data` operations based on the maximum and addition operations on weights.

Given an annotated RC-Tree, we locate a center by taking a path from the root down to the center based on the following observation. Consider two clusters C_1 and C_2 with a common boundary vertex v. Assume without loss of generality that $d_{C_1}(v) \geq d_{C_2}(v)$ and let u be a vertex of C_1 farthest from v. Then a center of $C_1 \cup C_2$ is in C_1, because any vertex in C_1 is no closer to u than v.

We find the center of the tree by starting at the root of the support tree and taking down a path to the center by visiting a cluster that contains a center next. To determine whether a cluster contains a center, we use the annotations to compute the distance from each boundary vertex to the rest of the tree and to the cluster and use the property of centers mentioned above.

3.6 Least Common Ancestors. Given a root r and vertices v and w, we find the least common ancestor of v and w with respect r by walking up the tree from all three vertices simultaneously until they all meet and then walking down two of the paths to find the least common ancestor.

Consider the cluster C that the paths from r, v, and w meet. If this is the first cluster that any two paths meet, then the vertex c joining into C is the least common ancestor. Otherwise, one of the children of C contains two clusters; move down to that cluster and follow the path down until the two paths split. If paths split at a binary cluster, proceed to the cluster pointing in the direction of the vertex whose path has joined last and follow the path down to the first unary cluster U; the vertex u joining into U is the least common ancestor. If the paths split at a unary cluster U, and both paths continue to unary clusters, then the vertex joining into U is the least common ancestor. Otherwise, continue to the binary cluster and follow the path down to the first unary cluster U; the vertex joining to U is the least common ancestor.

4 Implementation and the Experimental Setup

We implemented a library for dynamic trees based on change propagation as described by Acar *et al* [2] and implemented a general purpose interface for dynamic trees. The code for the library is publicly available at the home page of the Aladdin Center `http://www.aladdin.cs.cmu.edu`.

We performed an experimental evaluation of this implementation by considering a semi-dynamic minimum-spanning-tree algorithm, the max-flow algorithm of Sleator and Tarjan [17], and some synthetic benchmarks. The synthetic benchmarks start with a single primitive tree and its RC-Tree and apply a sequence operations consisting of links/cuts, data changes, and queries. Proper forests arise as a result of edge deletions.

This section describes the implementation, and how we generate the primitive input trees and the sequence of operations that our synthetic benchmarks rely on.

4.1 The Implementation. The implementation dynamizes a standard implementation of the tree contraction algorithm of Miller and Reif [14] by applying the dependence tracking techniques presented in other work [2]. The implementation represents each tree as a linked list of nodes ordered arbitrarily. Each node has an adjacency list of pointers to its neighbors in the tree. Dependence tracking requires remembering the complete tree from each round and creating dependences between copies of the same node between consecutive rounds. The dependences are created by a special *read* operation that reads the contents of a vertex, performs a rake or compress operation on the vertex, and copies the contents to the next round if the vertex remains alive. The implementation is otherwise identical to the implementation of the standard tree contraction algorithm. Since the contracted tree from each round is stored in the memory the implementation uses expected $O(n)$ space.

Change propagation is implemented by maintaining a first-in-first-out queue of vertices affected by the changes and rerunning the rake and compress operations of the original algorithm on the affected vertices until the queue becomes empty. The edge insertion and deletions initialize the change propagation queue by inserting the vertices involved in the insertion or deletion. During change propagation, additional vertices are inserted to the queue when a vertex that they read (has a dependence to) is written.

Since the implementation uses change propagation to update the RC-Trees, the structure (shape, layout) of the data structure depends only on the most current primitive tree. The data structure is therefore history-independent as defined my Micciancio [13, 2]. The data structure can be made weakly history-independent according to the stronger definition of history independence by Naor and Teague by using their memory allocator [15].

4.2 Generation of Input Forests. Our experiments with synthetic benchmarks take a single tree as input. To determine the effect that different trees might have on the running time, we generate trees based on the notion of *chain factor*. Given a chain factor f, $0 \leq f \leq 1$, we employ a tree-building algorithm that ensures that at least a fraction f of all vertices in a tree have degree two as long as $f \leq 1 - 2/n$, where n is the number of vertices in the tree. When the chain factor is zero, the algorithm generates random trees. When the chain factor is one, the algorithm generates a degenerate tree with only two leaves (all other vertices have degree two). In general, the trees become more unbalanced as the chain factor increases.

The tree-building algorithm takes as input the number of vertices n, the chain factor f and the bound d on the degree. The algorithm builds a tree in two phases. The first phase starts with an empty tree and grows a random tree with $r = \max(n - \lceil nf \rceil, 2)$ vertices by incrementally adding vertices to the current tree. To add a new vertex v to the current tree, the algorithm randomly chooses an existing vertex u with degree less than d, and inserts the edge (u, v). In the second phase, the algorithm adds the remaining $n - r$ vertices to the tree T obtained by the first phase. For the second phase, call each edge of the tree T a *super edge*. The second phase inserts the remaining vertices into T by creating $n - r$ new vertices and assigning each new vertex to a randomly chosen super edge. After all vertices are assigned, the algorithm splits each super edge (u, v)

with assigned vertices $v_1, \ldots, v_l$ into $l+1$ edges (u, v_1), (v_1, v_2), $\ldots (v_{l-1}, v_l)$, (v_l, v). Since all of the vertices added in the second phase of the construction have degree two, the algorithm ensures that at least a fraction f of all vertices have degree two, as long as $f \leq 1 - 2/n$.

Our experiments show that the performance of RC-Trees is relatively stable for primitive trees with varying chain factors, except for the degenerate tree with only two leaves. When measuring the effect of input size, we therefore generate trees with the same chain factor (but with different number of vertices). For these experiments, we fix the chain factor at 0.5. Since the data structure is stable for a wide range of chain factors, any other chain factor would work just as well.

For all our experiments that involve synthetic benchmarks, we use degree-four trees. We use degree-eight trees for the semi-dynamic minimum spanning trees, and the max-flow applications.

4.3 Generation of Operation Sequences. For the synthetic benchmarks, we generate a sequence of operations and report the time per operation averaged over 1024K operations. All operation sequences contain one of the following three types of operations

1. application-specific queries,
2. application-specific-data changes, and
3. edge insertions/deletions (link and cuts).

We generate all application-specific queries and data changes randomly. For queries, we randomly pick the vertices and edges involved in the query. For data changes, we pick random vertices and edges and change the labels (for vertices) and weights (for edges) to a randomly generated label or weight respectively.

For generating edge insertions/deletions, we use two different schemes. The first scheme, called *fixed-chain-factor scheme* ensures that the chain factor of the forest remains the same. Most experiments rely on this scheme for generating the operations. The second scheme, called *MST-scheme*, relies on minimum spanning trees, and is used for determining the performance for change propagation under batch changes.

— *Fixed-chain-factor scheme:* This scheme generates an alternating sequence of edge deletions and insertions by randomly selecting an edge e, deleting e, and inserting e back again, and repeating this process. Since the tree is restored after every pair of operations, this scheme ensures that the chain factor remains the same.

— *MST Scheme:* The MST scheme starts with an empty graph and its MST, and repeatedly inserts edges to the graph while maintaining its MST. To generate a sequence of operations, the scheme records every insertion into or deletion from the MST that arise as a result of insertions to the graph.

In particular, consider a graph G, and its minimum-spanning forest F, and insert an edge (u, v) into G to obtain G'. Let m be the heaviest edge m on the path from u to v. If the weight of m is less than the weight of e, then F is a minimum-spanning forest of G'. In this case, the forest remain unchanged and the scheme records nothing. If the weight of m is larger than that of e, then a minimum-spanning forest F' of G' is computed by deleting m from F and inserting e into F. In this case, the scheme records the deletion of m and insertion of e.

Since our implementation of RC-Trees assumes bounded-degree trees, we use a version of this scheme that is adapted to bounded-degree trees. If an inserted edge e causes an insertion that increases the degree of the forest beyond the predetermined constant, then that insertion is not performed and not recorded; the deletion, however, still takes place.

5 Experimental Results

We ran all our experiments on a machine with one Gigabytes of memory and an Intel Pentium-4, 2.4 GHz processor. The machine runs the Red Hat Linux 7.1 operating system.

5.1 Change Propagation. We measure the time for performing a change propagation with respect to a range of chain factors and a range of sizes. Figures 6 and 7 show the results for these experiments. To measure the cost of change propagation independent of application-specific data, this experiment performs no annotations—`-rake_data`, `compress_data`, and `finalize_data` operations are specified as "no-ops".

Figure 6 shows the timings for change propagation with trees of size 16K, 64K, 256K, 1024K with varying chain factors. Each data point is the time for change propagation after one cut or one link operation averaged over 1024K operations. The operations are generated by the fixed-chain-factor scheme. As the timings show, the time for change propagation increases as the chain factor increases. This is expected because the primitive tree becomes more unbalanced as the chain factor increases, which then results in a larger RC-Tree. When the chain factor is one, the primitive (degenerate) tree has only two leaves. Trees with only two leaves constitute the

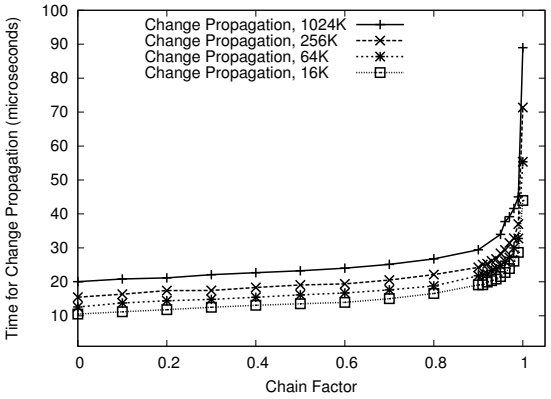


Figure 6: Change Propagation & chain factor.

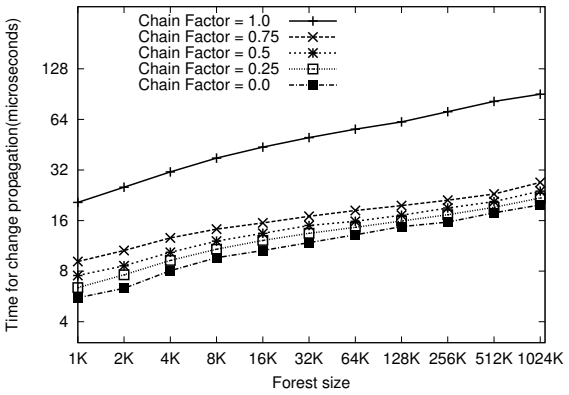


Figure 7: Change propagation & input size.

worst-case input for change propagation, because this is when the depth of the corresponding RC-Trees are (probabilistically) large compared to a more balanced tree of the same size. As additional data points for $0.91, 0.92, \ldots, 0.99$, show the steepest increase in the timings occurs when the chain factor increases from 0.99 to 1.0. This experiment shows that change-propagation algorithm is stable for a wide range chain factors for both small and larger trees.

Figure 7 shows the timings for varying sizes of trees with chain factors $0, 0.25, 0.50, 0.75$, and 1.0. Each data point is the time for change propagation after one link or cut operation averaged over 1024K operations. The operations are generated by the fixed-chain-factor scheme. The experiments show that the time for change propagation increases with the size of the input, and that the difference between trees of differing chain factors are relatively small, except for the degenerate tree (chain factor of one). This experiments suggests that the time for change-propagation is logarithmic in the size of the input and proven by previous work [2].

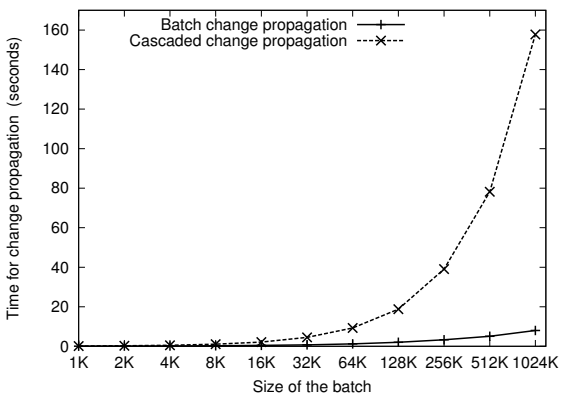


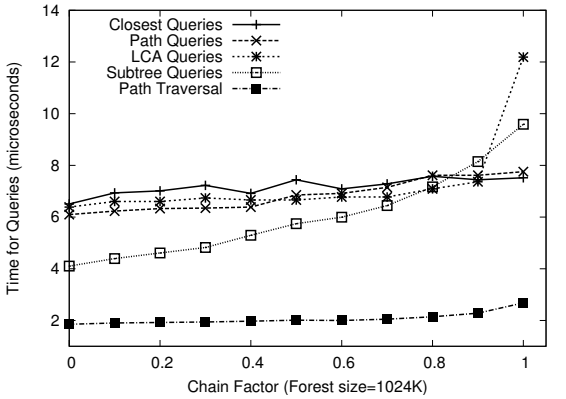
Figure 8: Batch change propagation.

5.2 Batch Change Propagation. An interesting property of change propagation is that it supports batch operations directly without any change to the implementation. Figure 8 shows the timings for change propagation with varying number cut and link operations in batch and in standard cascaded (one-by-one) fashion. The sequence of operations are generated by the MST scheme for a graph of 1024K vertices. The x axis (in logarithmic scale) is the size of each batch increasing from 1K to 1024K. Each data point represents the total time for processing the given number of cut and link operations in a batch or in a cascaded fashion.

The figure suggests a logarithmic factor asymptotic performance gap between processing changes in batch and in cascaded modes. This is expected, because, as the size of the batch approaches n, the change-propagation algorithm takes expected $O(n)$ time in the size of the input forest. Processing a x link or cut operations one-by-one, however, requires expected $O(x \log n)$ time.

5.3 Application-Specific Queries. Figure 9 shows the time for various application-specific queries for varying chain factors and input sizes. The queries for diameters and medians are not included in the figure, because they are very fast—they simply read the value at the root of the RC-Tree.

In these experiments, each data point is the time for one query averaged over 1024K randomly generated queries. For comparison the data line "path traversal" shows the time for starting at a leaf of the support tree and walking up to the root of that tree without doing any operations. The path-traversal time is measured by randomly selecting leaves and averaging over 1024K leaves. Since most interesting queries will at least walk up the RC-Tree (a standard constant-degree tree), the path traversal time can be viewed as the best possible



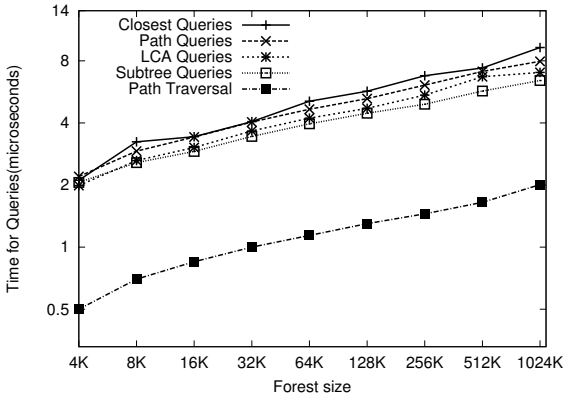
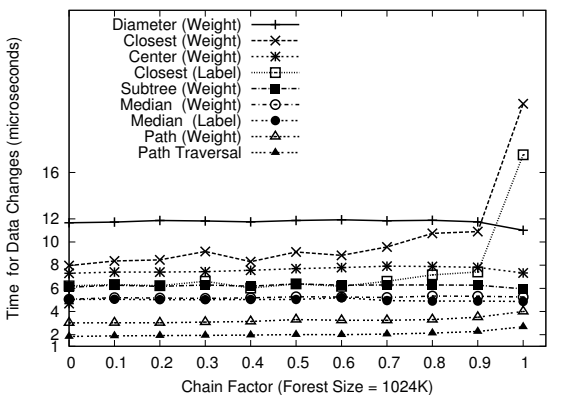


Figure 9: Queries versus chain factor and input size.



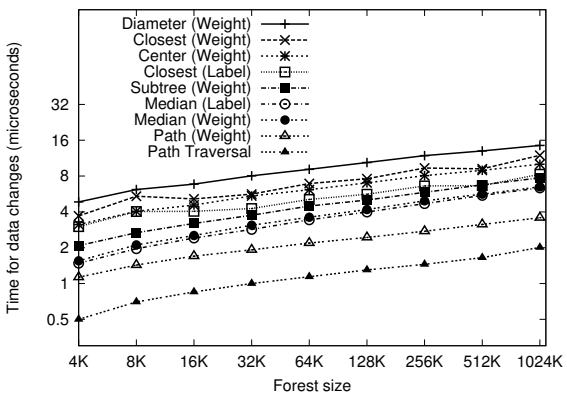


Figure 10: Weight changes vs chain factor & forest size.

for a query. As the figure shows, the time for all queries are within a factor of five of the path-traversal time.

The time differences between different queries is a result of the specific computations performed by each query. The least common ancestor (LCA) queries, for example, traverse three different paths bottom up until they meet and one path down. Subtree queries traverse two paths bottom up, and also touch the clusters neighboring the paths.

The timings suggest that all queries are logarithmic time. This is expected because all queries take time proportional to the height of the underlying RC-Tree.

5.4 Application-Specific Data Changes. We consider two types of application-specific data changes: label changes (for vertices), and weight changes (for edges). Processing a data change involves changing the weight or the label in the primitive tree and updating the RC-Tree with respect to the change. Since the weights and labels do not affect the structure of the RC-Trees, handling such changes require updating just the annotations of the RC-Trees. Data changes are therefore handled by changing the tag of the base cluster corresponding to the change, and updating the tags of all clusters on the path to the root.

Figure 10 shows the timing for edge weight and vertex label changes versus the chain factor and the size of the primitive forests. All our applications, except but for the Least-Common Ancestors, involve weighted trees. Weight changes are therefore relevant to all these applications. Vertex label changes however are only relevant to nearest-marked vertex, and the median applications. In the nearest-marked-vertex application, a label change can change an unmarked vertex to a marked vertex or vice versa. The mark indicates if that vertex is in the special set to which the distance of a given query node is measured. In the median application, a label change can change the contribution of a vertex to the weighted median.

As Figure 10 shows the time for data changes are relatively stable across a range of chain factors. For comparisons purposes, the figure also shows the time for path traversals. This measure is taken by randomly selecting leaves in the RC-Tree and measuring the time for traversing the path to the root (averaged over 1024K leaves). Since each data update traverses a path from

a leaf to the root of the tree, the path-traversal time can be viewed as a lower bound for processing a data change. Figure 10 shows that the time for data changes is within an order of magnitude of the time for path traversals for most applications except for the diameter application. In general, if the application involves complex annotations, then it is slower to process a weight or label change.

The timings suggest that all data changes take logarithmic time to process. This is expected, because RC-Trees have logarithmic height in expectation.

5.5 Comparison to Link-Cut Trees. We implemented the Link-Cut Tree interface of Sleator and Tarjan [17, 18] as an application to our library. The interface supports link and cut operations, path queries, and an *addCost* operation for adding weights to paths. We compare our implementation to Renato Werneck's [21] implementation of Link-Cut Trees using splay trees.

Figure 11 shows the timing for link and cut operations. As the figure shows, Link-Cut Trees (written LC-Trees in figures) are up to a factor of five faster than the RC-Trees for links and cuts.

Figure 12 shows the timings for data changes. For operations that add a weight to a path, Link-Cut Trees are up to 50% faster than RC-Trees. For operations that change the weight of one edge, RC-Trees are up to 50% faster than Link-Cut Trees.

Figure 13 shows the timings for queries. For path queries that ask for the heaviest edge on a path, RC-Trees are up to 60% faster that Link-Cut Trees. For comparisons purposes, Figure 13 also shows the timing for path queries, for a version of our interface that does not support adding weights on paths. These queries are up to a factor of three faster than Link-Cut Trees. Certain applications of path queries, such as Minimum-Spanning Trees, do not require adding weights on paths.

These comparisons show an interesting trade-off. Whereas RC-Trees tend to perform better in application-specific queries and data changes, LC-Trees perform better for structural changes (link and cut operations). As the experiments with minimum-spanning trees, and max-flow algorithms show, this trade-off between structural and data changes makes one data structure preferable to the other depending on the ratio of link-cut operations to queries. Note also that LC-Trees support path queries only and must be extended internally to support other type of queries such as subtree queries [5, 16] that RC-Trees support.

5.6 Semi-Dynamic Minimum Spanning Trees. We compare the performance of the implementations of a semi-dynamic minimum spanning tree algorithm

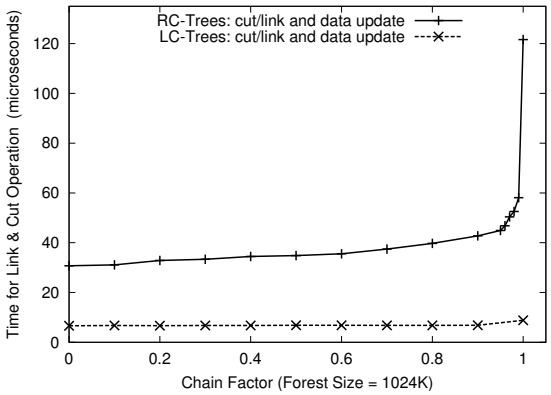


Figure 11: Link and cut operations.

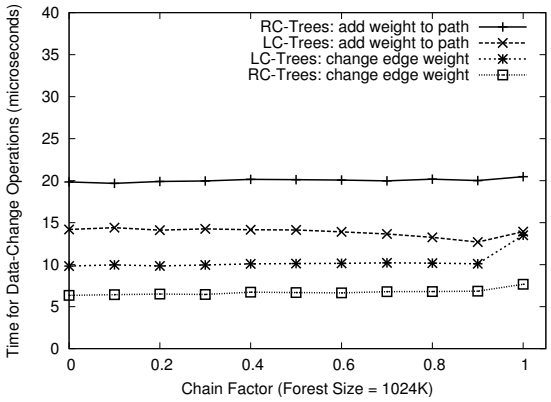


Figure 12: Data Changes.

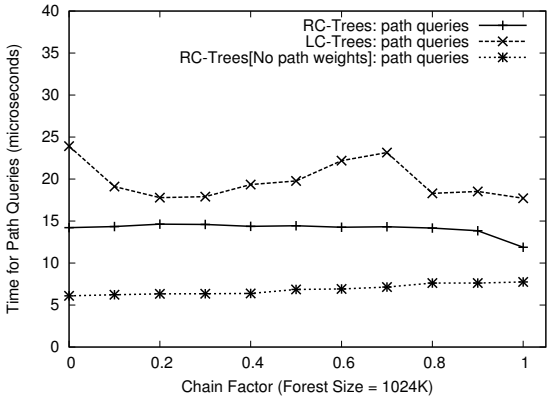


Figure 13: Path queries.

using RC-Trees and Link-Cut Trees. The semi-dynamic algorithm maintains the minimum-spanning tree of a graph under edge insertions to the graph. When an edge e is inserted, the algorithm finds the maximum edge m on the path between the two end-points of e via a path query. If the weight of e is larger than that

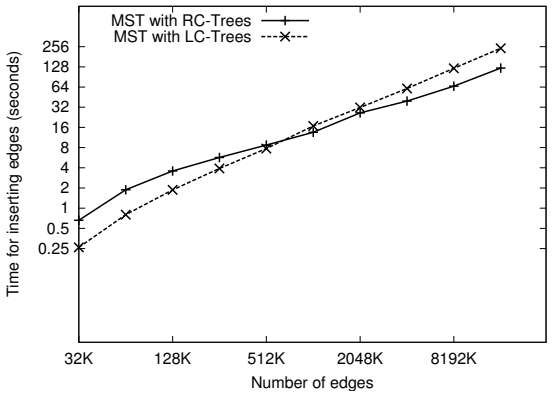


Figure 14: Semi-dynamic MST.

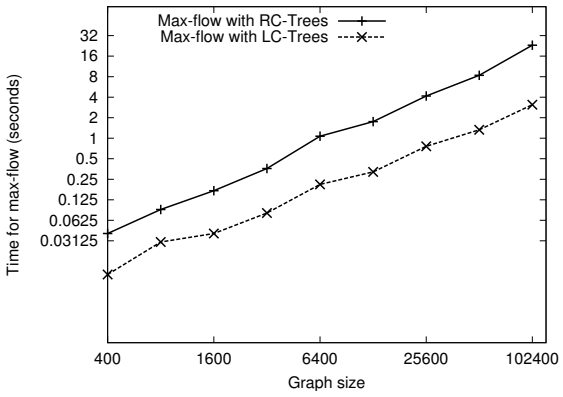


Figure 15: Max-flow

of m, the algorithm replaces m with e in the MST. If the weight of e is larger than that of m, then the MST remains the same. To find the maximum edge on a path quickly, the algorithm uses a dynamic-tree data structure to represent the MST; this enables performing path queries in logarithmic time. An edge replacement requires one cut and one link operation on the dynamic-trees data structure.

For this experiments, we compare two implementations of the semi-dynamic MST algorithm that only differ in their choice of the dynamic-trees data structure. We start with a graph of 32K vertices and no edges and randomly insert edges to the graph while updating its MST. We generate the sequence of insertions offline by randomly selecting a new edge to insert, and by randomly generating a weight for that edge. Since our implementation of RC-Trees supports constant-degree trees only, we generate a sequence of insertions that ensures that the underlying MST has constant degree. For this experiment, we used degree-eight RC-trees.

Figure 14 shows the timings using RC-Trees and Link-Cut Trees. Since the edge weights are randomly generated, insertions are less likely to cause replacements as the graph becomes denser. Therefore the number of queries relative to the number of link and cut operations increase with the number inserted edges. Since path queries with RC-Trees are faster than with Link-Cut Trees, the semi-dynamic MST algorithm performs better with RC-Trees for dense graphs.

5.7 Max-Flow Algorithms. As another application, we implemented the max-flow algorithm of Sleator and Tarjan [17]. The algorithm uses dynamic trees to find blocking flows. For the experiments, we used the DIMACS's Washington Generator in *fct 2* mode to generate *random level graphs* with 5 rows and $n/5$ columns, with capacity up to 1024K. The sum of the in-degree and the out-degree of any vertex in a random level graphs generated with these parameter is bounded by 8. We therefore use degree-eight RC-Trees for this experiment.

A random level graphs with r rows and l levels consists of $rl + 2$ vertices. The vertices consists of a source, a sink, and r rows. Each row consists of l vertices, one vertex for each level. A vertex at level $1 \leq i < l$ is connected to three randomly chosen vertices at level $i + 1$ by an edge whose capacity is determined randomly. In addition, the source u, is connected to each vertex at the first level by an edge, and each vertex at level l is connected to the sink v by an edge. The capacities of the edges incident to the source and sink are large enough to accommodate any flow.

As Figure 15 shows the timings for two implementation of the max-flow algorithm of Sleator and Tarjan that differ only in their choice of the dynamic-tree data structure being used. As the timings show the LC-Trees implementation is faster than the RC-Trees implementation. This is expected because the algorithm performs more structural changes (link and cuts) than queries.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[2] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 524–533, 2004.

[3] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 270–280, 1997.

[4] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully-dynamic trees with top trees, 2003. The Computing Research Repository (CoRR)[cs.DS/0310065].

[5] R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–61, 1991.

[6] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[7] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal of Computing*, 26:484–538, 1997.

[8] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal Algorithms*, 24(1):37–65, 1997.

[9] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[10] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *SIAM Journal of Computing*, 28(2):612–636, 1998.

[11] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 46(4):502–516, 1999.

[12] J. Holm, K. de Lichtenberg, and M. Thorup. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

[13] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 456–464, 1997.

[14] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[15] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 492–501. ACM Press, 2001.

[16] T. Radzik. Implementation of dynamic trees with in-subtree operations. *ACM Journal of Experimental Algorithms*, 3:9, 1998.

[17] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[18] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[19] R. Tarjan and R. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[20] R. E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:167–177, 1997.

[21] R. Werneck. Princeton University.