Dynamic Trees through Euler Tours
A purely functional programming approach

Juan Carlos Saenz-Carrasco* and Mike Stannett

Department of Computer Science,
Regent Court, 211 Portobello,
Sheffield S1 4DP, United Kingdom
{jcsaenzcarrasco1,m.stannett}@sheffield.ac.uk
https://www.sheffield.ac.uk/

Abstract. We present purely-functional data structures for managing dynamic updates and queries. By doing so, we combine the structure of the Hinze's and Paterson finger tree with an efficient binary search tree to handle Euler tours as sequences: like its imperative counterpart, our proposal supports $O(\log n)$ amortised per operation, where n is the number of the vertices in the tree. While existing data structures provide these time bounds, none do so to provide the three basic operations on dynamic trees: link, cut and connected altogether within the same functional structure. However, in the nearly twenty years since, no one has shown that these functions work efficient under persistent data structures. *Some efforts have been done, for instance chahine.moreau@gmail.com at https://github.com/k0ral/euler-tour-tree. However there no proofs about correctness or efficiency. Also the part corresponding to forests is missing* To understand how the algorithms perform in practice we have also implemented them in the Haskell functional programming language. Our implementations are not as fast as the best of these on the ephemeral and imperative realm but comparable by the same order of magnitude.

Keywords: Functional programming, persistent data structures, Haskell, *Experimental results*

1 Introduction

In this paper we consider whether functional programming features (for example, the use of lazy evaluation) can be exploited in the context of massive dynamically changing networks. Such networks—the network of hyperlinked pages constituting the World Wide Web, for example, or the emerging Internet of Things (IoT)—are increasingly ubiquitous and increasingly important to modern commerce and society. With growing size, however, come additional computational

* Postgraduate research student, supported by the Consejo Nacional de Ciencia y Tecnología, CONACYT (the Mexican National Council for Science and Technology) under grant 411550, scholar 580617 and CVU 214885

overheads, and it is important to determine whether low-complexity, preferably sub-linear, algorithms can be derived for such key tasks as: finding paths between nodes; re-routing network traffic due to local outages; and distributing workloads strategically to maximize network efficiency.

In general, the number of vertices in such a network can change from one moment to the next, but if we restrict ourselves to any given finite period of time the total number of vertices involved (including vertices that may at some point have been deleted) will be finite. Similarly, the total number of connected components within the network may change from time to time, as new edges are added and old ones deleted. Nonetheless, at any given time there will be some definite number of such components, and we are free to represent each component in compressed form using one of its spanning trees.

In this paper, then, we start by assuming the existence of some set V of n vertices (n constant), which are connected by a dynamically changing set of edges to form a forest (i.e. a collection of trees). The problem we address is one of the simplest we can ask about a network, but at the same time one of the most fundamental, namely: *given two vertices, u and v, do they belong to the same component*? The answer, of course, will depend on when we ask the question, because the system we're considering is *dynamic*; new connections can be created between vertices and old ones can be deleted.

Given our use of spanning trees to represent connected components, this question becomes: *do u and v belong to the same tree*? Notice that adding new links within an already connected component does not affect its existing spanning trees; although new spanning trees may become possible, no existing spanning tree loses that status. On the other hand, if we connect vertices from distinct components, joining those nodes in the associated spanning trees automatically creates a new spanning tree for the larger component created by adding the link. (The relationship between deleting an edge in a component vs. deleting it in its spanning tree is more complex–see Sect. **??**.) We accordingly assume that the forest can change dynamically in response to repeated applications of two basic operations: link and cut.

- if vertices u and v belong to different trees, $\mathsf{link}(u, v)$ adds the edge (u, v) to the forest, thereby causing the trees containing u and v to be joined together to form a new, larger, tree; if the vertices already belong to the same tree, the operation has no effect.
- if vertices u and v are connected by an edge, $\mathsf{cut}(u, v)$ removes that edge, thereby causing the tree that contains it to be split into two smaller trees; if the vertices are not connected, the operation has no effect.

WORKING HERE

The problem has been studied as *dynamic trees* under different variants according to the application, for instance, Sleator and Tarjan defined the *link-cut* tree for solving network flow problems [12], Henzinger and King provided *Euler tour* (ET) trees for speeding up dynamic connectivity on graphs [8] or Frederickson describing the *topology* tree for maintaining the minimum spanning tree [5]. The former and latter trees are based on techniques called *path-decomposition*

and *tree contraction* respectively. In this paper we are interested on Euler tour trees, a technique called *linearisation*, since the original tree (of any degree) is flattened and handled it as a sequence, in other words, turning a non-linear structure into a linear one.

All the above techniques have been implemented and studied for ephemeral data structures, performing $O(\log n)$ per operation. Functional data structures, on the other hand, ease the reasoning and implementation for the cases when preserving history is needed as in self-adjusting computation [1] or as in version control [3]. To overcome the lack of pointers, researchers have devised data structures to represent efficient sequences, most notably, *finger trees* [10]. This structure performs well, allowing updates and queries in logarithmic time. We introduce the Euler-tour-finger-tree, a variant of the Hinze's and Paterson finger trees, or ETFT for short. Like the finger tree, the ETFT edit sequences in logarithmic time while providing dynamic tree operations. The key insight is to make $O(1)$ steps to perform *connected*, *link*, and *cut* in logarithmic time. *NO FURTHER explanation about the $O(1)$ performance*

The contribution of this paper is to give first work-optimal bounds for managing dynamic trees for purely functional data structures. We do this not only for extending the application of finger trees, we offer a solution for dynamic connectivity when facing persistent data structures as well as providing a simple interface to the user. The ETFT exposes two types, `Tree` and `Forest`, to the user. The following is a compilation of *OUR* the functions, types and times we have implemented in Haskell, on top of that done by Hinze and Paterson [10]:

Function	Type	Time complexity
`connected`	`Vertex → Vertex → Forest → (Bool, PairTreeVertex)`	$O(\log n)$
`linkTree`	`Vertex → Tree → Vertex → Tree → Tree`	$O(\log n)$
`cutTree`	`Vertex → Vertex → Tree → (Tree,Tree)`	$O(\log n)$
`link`	`Vertex → Vertex → Forest → Forest`	$O(\log n)$
`cut`	`Vertex → Vertex → Forest → Forest`	$O(\log n)$
`reroot`	`Tree → Vertex → Tree`	$O(\log n)$
`root`	`Tree → Maybe(Vertex)`	$O(1)$

The first block (first three rows) contains the functions to perform dynamic tree operations. The following block (rows 6 and 7 4 and 5) lists the functions that compute an unbounded, although finite, sequence of dynamic tree operations over the same forest F; the suffix `Rec` represents recursion. Finally, last two rows are the core functions, apart from the ones provided by Hinze and Paterson work [10], to perform dynamic tree operations. *PERHAPS explain the above table widely or paraphrasing ???*

In the next section, we give an overview of the finger tree functions and their performance. These are the foundations for our functions altogether with some of the functions from the Hackage library `Data.Set` [1] which mimic the set-like operations over our functions, such as testing membership.

[1] https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Set.html

In Section **??**, we present an in-depth example using the operations mentioned above. In particular, we depict how the ETFT manages the trees as sequences under the dynamic setting, showing how the ETFT structure is constructed immutably. In Section 3, we describe our implementation for performing dynamic tree operations with purely data structures, the ETFT. The following is the address where our source code is hosted and publicly available:

`https://github.com/jcsaenzcarrasco/dynTs`

We evaluate ETFT empirically in Section 4. In concrete, we plot the results of benchmarking the functions in the first two sections of the table above. Our evaluation demonstrates that ETFT performs according the function definitions with a constant factor between `connected` and `link` and `cut`.

In Section 5 we suggest potential alternatives for designing dynamic data structures to solve the problem described in the present document.

Finally, we conclude in Section 6 with a summary of our work.

Presentationally I'm missing a better description of the background material. The paper recalls the definition of a monoid, but does not recall the definition of an Euler tour nor give enough detail on finger trees to read as an independent document.

——— - P O S I T I V E C O M M E N T S ———

This paper introduces Euler Tour Finger Trees (ETFTs), a persistent, purely-functional tree and forest data structure that supports O(log n) complexity for all operations, including link and cut. It gives an implementation in Haskell.

At a high-level, the tree is represented as an Euler Tour (a list of vertex pairs describing a path through the tree), which in turn is represented by a finger tree, thereby allowing O(log n) concatenation and splitting. That allows manipulating Euler tours efficiently, and building interesting tree operations easily.

Most of the implementation in this paper thus builds on existing implementations of finger trees and sets. Yet the combination seems novel, and the complexity bounds of the ETFT operations follow almost trivially, which is nice. The paper presents a functional data structure for maintaining a dynamic forest under link and cut operations. It realizes this in O(log n) time complexity per operation (amortized), using an Euler tour tree representation implemented using a combination of finger trees and sets.

The paper first recalls dynamic trees and their operations, and then monoids, sets implemented with balanced search trees, Euler-tour trees and finger trees. It then gives (graphical) examples of how the sets and finger trees fit together to realize dynamic trees, it recalls graphically the root, reroot, split, concatenation, and view operations of finger trees, and then walks through the implementation of root, reroot, connected, link, and cut. Finally the paper contains experiments illustrating that the implementation meets the claimed complexity bounds and discusses related and future work. This is an interesting piece of work.

2 Preliminaries

Our work and contribution relies mainly on three structures, a *monoid*, a *set* and a *finger tree*, which are briefly described in the following subsections.

2.1 Monoid

A *monoid* is a triple $(S, \star, e)$, where S is a set, $\star$ is a binary operation, called *product* and e is an element of S, called *unit*, satisfying the following properties:

1. $e \star x = x = x \star e$, for all $x \in S$
2. $x \star (y \star z) = (x \star y) \star z$, for all $x, y, z \in S$.

As an example of a Haskell implementation of a monoid we have:

```
1 class Monoid  a where
2   mempty  :: a
3   mappend :: a → a → a
```

where the function `mempty` represents the element e and the function `mappend` represents function $\star$. Detailed information about monoids within the functional programming can be found in [14], and for the Haskell implementation at [6].

2.2 Set as binary search tree

A *binary search* tree (BST) is either a *leaf* (also called a `tip`) or a vertex consisting of a *value*, a *left* BST and a *right* BST. The height of the tree determines the time taken to perform every operation onto it, therefore the shorter the height the better, and this is done throughout a *balancing scheme*. The study of BSTs is vast and there are several implementation around. In our case, a *size balanced* is used. A detailed comparison and benchmarking is outside the scope of the present work.

My main criticism of the paper is the presentation. While I appreciate the use of diagrams in the first half of the paper, high-level explanations are missing later. The paper does not do a very good job providing an intuition for how the data structure works. For example, I was struggling understanding the role of the monoidal set. The paper also explains too little of the background regarding used existing data structures and their realisation in Haskell libraries. For example, the interface to the finger tree ADT and the Measured type class. In general, most of the explanations are closely tight to the code but dont give a high-level picture of what individual operations do in terms of the Euler tour. Given that there is plenty of space left in the paper, this could be improved in the final version.

The following is an snippet of `Data.Set` [2]

[2] https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Set.html

```
data Set a    = Bin Size a (Set a) (Set a)
              | Tip
```

In Table 1 we show the set-functions we have incorporated in `dynTsET` from `Data.Set`.

Function	Type	Time complexity
`empty`	`Set a`	$O(1)$
`insert`	`a → Set a → Set a`	$O(\log n)$
`member`	`a → Set a → Bool`	$O(\log n)$
`union`	`Set a → Set a → Set a`	$O(m(\log \frac{n}{m} + 1))$

Table 1. Leijen's implementation of `Data.Set` [11], based on [2]

Functions *empty* and `union` are, in fact, the monoidal functions `mempty` and `mappend` respectively.

2.3 Euler-tour tree

Dealing with trees of different degree can be complicated. A simple way to handle and represent trees of any degree is by an Euler tour, that is, a sequence as in [8] and [13]. To represent a tree t, we replace every edge $\langle u, v\rangle$ of t by two arcs (u, v) and (v, u), and add a loop (v, v) to represent each vertex v. In this context, a tree t can have at least one, and in general, many Euler tours. The size of an Euler tour et of t is $et(t) = v + 2e$, where v is the number of vertices of t and e its number of edges. We can represent an Euler tour in Haskell simply as a list of pairs, such in

```
data EulerTour a = [(a,a)]
```

By managing the tour with lists, we can perform insertion from the left (head) in $O(1)$ but remaining operations such insertion from the right, access, cutting, appending and inserting might take $O(n)$ per operation. On the other hand, representing tours through finger trees, performance per operation is improved up to $O(\log n)$ per operation amortised as we explain shortly and in Section 3.

Our representation of k-trees through Euler-tour will not close up the tour with the first node as this avoids the uniqueness presence for such a node, as shown in Fig. 1.

2.4 Finger tree

We present Hinze and Paterson's version of finger trees (FTs) [10], followed by the functions we have used for the `dynTsET`.

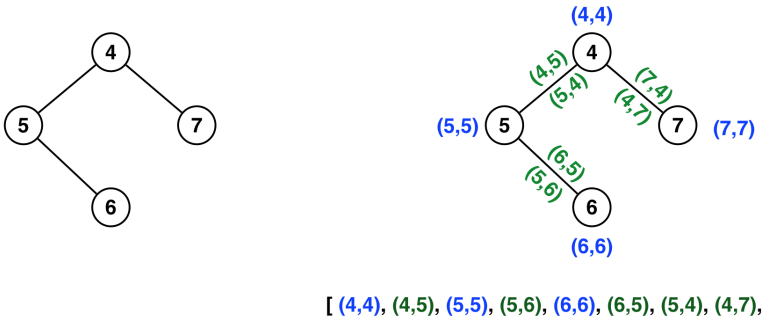

Fig. 1. The k-tree (left) is represented as a sequence of size $v + 2e$ (right). Notice we left out the final node-pair to preserve uniqueness

```
1 data FingerTree v a = Empty
2                     | Single a
3                     | Deep v
4                            Digit a
5                            FingerTree v (Node v a)
6                            Digit a
```

This reviewer can't help but think that there seems to be a lot of redundancy in the proposed data structure, as the Euler tour node pairs are present in both the sets and in the finger trees.

`Digit` type holds from one up to four elements of type `a`. `Node` type can hold two or three elements of type `a`. The recursive and nested definition of `FingerTree` forces the structure to be balanced by its types, instead of enforcing it by code invariants. Finger trees are 2-3 trees where values are stored at the leaves, in our case those leaves are the pairs representing an Euler tour. To implement *updates* and *lookups* efficiently, Hinze and Paterson [10] added a monoidal annotation on the intermediate vertices, the `v` type.

In figure Fig. 2 we can see our example of the Euler-tour sequence in Fig. 1 managed by the data constructors described above.

SPLIT is not longer used, instead there is SEARCHFT, which aim is the same as SPLIT and hope it will be easier to explain

Is it the "best" part in the document to bring up an explanations or examples or both for amortisation?

3 ETFT Design

Once a tree t of any degree is turned into an Euler tour et, we are able to manage et as a sequence. Since finger trees work efficiently on sequences we just need a few steps to *link* and *cut* trees by splitting and concatenating sequences, which

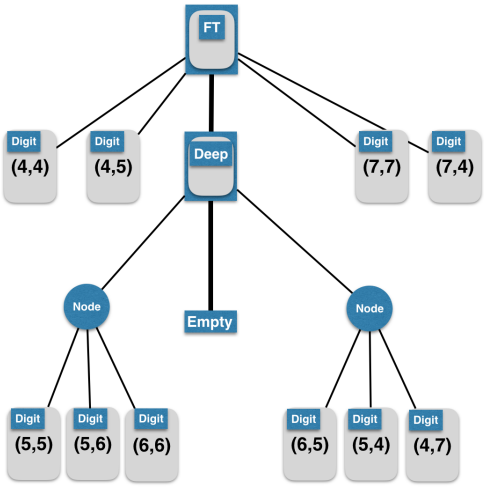

Fig. 2. A finger tree (FT) holding the sequence (Euler-tour) that represents the k-tree in Fig.1

take logarithmic time to compute them. Additional functions *cons*, inserting to the front of the sequence and *snoc*, inserting to the end of the sequence, just take $O(1)$, according to Table **??**.

Recall that functions in Table **??** work with finger trees of any type `a` constrained to `v` monoidal annotation. In order to manipulate vertices and edges as Euler tours, we specialized the ETFT types to handle pairs, hence the underlying finger tree types are `Set (a,a)` for the monoidal annotation (previously `v`) and `(a,a)` for values on the leaves (previously `a`).

The full implementation of ETFT[3] contains a simplified version of that in [9], so we can reason with the type constructors from the finger trees into the ETFT functions.

SUBSECTION FOREST. Either here or before section 4.1

For practical purposes we will show a forest and trees holding integers in this paper, but a polymorphic type can be extended easily.

We start with type definitions for vertices, trees, forests, and their empty instances as in Figure 3

Every element in the tree t (leaves in the finger tree) is measurable by calling `measure`, which returns the set-insertion operation of the element (v, u) into an empty set. Here `Data.Set` is imported qualified as `S`

The *root* of a tree t returns a vertex v or nothing if t is empty:

```
1 root :: FTInt → Maybe Vertex
2 root tree = case viewl tree of
3   NoView   → Nothing
```

[3] https://github.com/jcsaenzcarrasco/ETFT

```
1  type FTInt  = FT (S.Set (Int,Int)) (Int,Int)
2  type Forest = FT (S.Set (Int,Int)) FTInt
3  type Vertex = Int
4
5  emptyTree :: FTInt
6  emptyTree =  Empty
7
8  emptyForest :: Forest
9  emptyForest =  Empty
10
11 instance (Ord a) ⇒ Measured (a,a) (S.Set (a,a)) where
12    measure (x,y) = S.insert (x, y) S.empty
```

Fig. 3. code for `tree`,`forest`,`vertex` definitions and their empty cases

```
4   View x _ → Just $ fst x
```

Time complexity of `root` is $O(1)$ since `viewl` performs $O(1)$ and we simply pattern matching over its results.

Then, *rerooting* a tree t according to vertex v returns the same tree t on which v is the root, i.e. the first element in the sequence. This is useful when linking and cutting trees. Rerooting an empty or single tree is trivial (lines 3 and 4, Figure 4). Lines 6 and 7 show the cases that v is not in t or is already the root,

```
1  reroot :: FTInt → Vertex → FTInt
2  reroot tree vertex = case tree of
3   Empty       → Empty
4   (Single x)  → Single x
5   tree        → case split (S.member root) S.empty tree of
6                   NoSplit           → tree
7                   Split Empty _  _  → tree
8                   Split tl    _  tr → let (View _ tX) = viewl tl
9                                           tA          = tX ▷ root
10                                          tB          = root ◁ tr
11                                      in  tB ⋈ tA
12  where root = (vertex,vertex)
```

Fig. 4. code for *reroot* function

respectively in Figure 4. From the left subtree of the split (line 8), we simply ignore the first element derived from its `viewl` which is the old root, then we simply concatenate the tail and the head in that order altogether the new root inserted in both sides. That is, one *split*, one *concatenation* and two insertions *cons,snoc*, turns to be $O(\log n)$ in the size of the tree t.

3.1 connected

It is the first of our dynamic tree operations and the core function for *link* and *cut*. It receives two vertices u and v and a forest f as arguments, returning a boolean altogether with a tree or trees and their roots where applicable. It is based on a single *split* per vertex. If the split succeeds, then a tree and its root are returned, otherwise nothing is returned. This initial step is called `search` and takes as much as $O(\log n)$ per split and $O(1)$ for the *root*. Then `connected` pattern matches all the cases from `search` and compares whether or not the given vertices are in the same tree.

```
1  search :: Vertex → Forest → Maybe (FTInt, Vertex)
2  search v f =
3   case split (S.member (v,v)) S.empty f of
4    NoSplit        → Nothing
5    Split _ tree _ → Just (tree, fromJust $ root tree)
6
7  connected :: Vertex → Vertex → Forest → (Bool, Maybe PairTreeVertex)
8  connected x y f =
9   case (search x f, search y f) of
10   (Nothing     , _           ) → (False, Nothing)
11   (_           , Nothing     ) → (False, Nothing)
12   (Just (tx,rx), Just (ty,ry)) → if rx == ry
13                                    then (True,  Just(tx,rx,tx,rx))
14                                    else (False, Just(tx,rx,ty,ry))
```

Fig. 5. the *connected* function, the basic query for dynamic trees and the core for *link* and *cut*

Following lines 9-14 in Figure 5, the first two cases of `connected` occur when any of u or v are not members of f. Last two lines handle the restrictions for functions *cut* and *link* respectively. That is, if two vertices u and v are connected, then both have the same root of the same tree, preserving the *cut* condition. The last line states the valid case for *link*, that is, not connected u and v that belong to f, therefore `connected` also returns the corresponding trees and roots. Finally, we conclude that `connected` takes as much effort as `search`, $O(\log n)$, where n is the number of vertices in the forest.

3.2 link

Performing a *link* implies that given two vertices u and v and a forest f, u and v should belong to a different trees t_u and t_v in f. This condition is guarded by `connected`, in case of non existent u or v and no connection between, f is simply returned. This will be useful when `link` is part of the unbound sequence of operations applied to a forest. The *link* condition is stated and satisfied in

the case analysis, lines 3-5 in Figure 6. Now, we need to reduce the forest f by one tree as t_u and t_v will be joined. Hence t_u and t_v are extracted from f and dropped off (left apart with `_`), lines 7 and 8 in Figure 6. Finally, `linkAll` joins subforests `lf`,`rf` and inserts the new tree (`tree`) resulting from auxiliary function `linkTree`.

```
1 link :: Vertex → Vertex → Forest → Forest
2 link x y f =
3  case connected x y f of
4   (False, Just (tx,rx,ty,ry)) → linkAll (linkTree x tx y ty)
5   _                           → f
6  where
7     Split lf' _ rf' = split (S.member (x,x)) S.empty f
8     Split lf  _ rf  = split (S.member (y,y)) S.empty (lf' ⋈ rf')
9     linkAll tree    = tree ◁ (lf ⋈ rf)
```

Fig. 6. the *link* function, reduces the size of the forest by one tree

Inserting the new tree into the new forest takes $O(1)$ due `cons`, but it is the computation of `linkTree` that takes $O(\log m)$, where m is the size of the trees t_u and t_v as it applies two *reroot* and a concatenation, lines 4-6 in Figure 7. Finally, joining the subforests derived from eliminating trees t_u and t_v takes three splits, lines 3, 7 and 8, which take $O(\log n)$ each and a concatenation, line 9 which also takes $O(\log n)$, where n is the number of vertices in the forest f. In the worst case, if f is formed by a single tree t after linking, then n and m are equivalent.

```
1 linkTree :: Vertex → FTInt → Vertex → FTInt → FTInt
2 linkTree u tu v tv =
3    let
4       from = (reroot tu u ) ▷ (u,v)
5       to   = (reroot tv v ) ▷ (v,u)
6    in (from ⋈ to ) ▷ (u,u)
```

Fig. 7. Auxiliar *linkTree* function, the *join* between two different trees

3.3 cut

Unlike *link*, the function *cut* adds a new tree into the forest f. The condition for *cut* to be valid, is that input vertices u and v in forest f should also belong the same tree t. We also add the condition that a vertex can not be cut, just edges. The latter condition is guarded in line 3 in Figure 8 whereas former condition

is preserved by pattern matching in lines 6 and 7 within the same figure. Like *link*, function *cut* will return the original forest f if any of these conditions are not satisfied. Again, this helps the unbounded sequence of dynamic operations to be applied over f.

I miss a deeper discussion of the interplay of this use of the finger tree and the monoidal annotations. For example, the 'reroot' operation should leave the set of edges invariant. Do the annotation on the top node have to be recalculated? Can it be avoided? Would it make a difference?

```
1  cut :: Vertex → Vertex → Forest → Forest
2  cut x y f
3   | x == y    = f
4   | otherwise =
5     case connected x y f of
6       (True, Just (tx,_,_,_)) → buildForest (cutTree x y tx)
7       _                       → f
8   where
9     buildForest (t2,t3) = t2 ◁ (t3 ◁ (lf ⋈ rf))
10    Split lf _ rf       = split (S.member (x,x)) S.empty f
```

Fig. 8. The *cut* function increases the size of the forest f by one tree

If the *cut* function is successful, then tree t is splitted into trees t_u and t_v according the input vertices u and v, hence t_u is the tree containing vertex u but not v and t_v is the tree containing vertex v but not u, denoted as trees `t2` and `t3` in Figure 8 respectively. Also, we need to remove t from forest f. The insertion of t_u and t_v into f and removal t from f is done in lines 9 and 10. Thus, function *cut* takes two splits, one from `connected` in line 5, and from line 10, which turns in $O(\log n)$. Also, a concatenation is performed in rebuilding the forest f, in line 9, adding up another $O(\log n)$, where n is the number of vertices in f. Additionally to this, auxiliary function `cutTree`, pictured in Figure 9, takes one viewing (`viewr`) which is $O(1)$, then, one rerooting, two splits and one concatenation, each of $O(\log m)$, where m is the number of elements in t. In the worst case, if f contains a single tree t before cutting, then m and n are equivalent.

4 Evaluation

In this section we evaluate ETFT. We demonstrate that *link*, *cut* and `connected` are efficient, that is, they have amortised complexity $O(\log n)$. We compiled our code through `ghc` verion 8.0.1, and ran it on a 2.2 GHz Intel Core i7 MacBook Pro with 16 GB 1600 MHz DDR3 running macOS High Sierra version 10.13.1 (17B1003). We use the [9] code for finger trees.

THIS SECTION IS TINY

```
1 cutTree :: Vertex → Vertex → FTInt → (FTInt,FTInt)
2 cutTree u v tree = case split (S.member (u,v)) S.empty tree of
3  NoSplit → (tree,Empty)
4  _       →
5   let treeU          = reroot tree u
6       Split  treeA' _ right = split (S.member (u,v)) S.empty treeU
7       View _ treeA          = viewr treeA'
8       Split treeB _ treeC   = split (S.member (v,u)) S.empty right
9   in  (treeB, treeA ⋈ treeC)
```

Fig. 9. Auxiliary function *cutTree splits* the given tree in two

Experimental Setup. We present the following experiment. We construct a forest from scratch by inserting elements, through `link` in descending order, followed by queries through `connect` to the just created forest, and then split those trees with`cut`. Upon reaching a target length, we record the total time taken. We use target lengths of 50K to 500M elements (vertices), and repeat the process for a total of five data points for each target. We plot the median of the results, shown in Figure 10.

Deduce regression line in Fig 10

Results. As expected, the growth of the curves by *link* and *cut* is linear. Given a sequence of size m of update operations which take $O(\log n)$ each, the overall time turns into $O(m(\log n))$. This is because every update operation changes the size of the forest which is carry on all the way in the sequence. Finally, *connected*, takes almost constant time, in fact $O(\log n)$, since the size of the forest remains fixed during the sequence.

The introduction promises O(log n) amortised runtimes, but the paper never discusses the amortised aspect. Amortised over what usage? Similarly, the introduction mentions a "key insight" of O(1) operations, but this train of thought is likewise not picked up later.

Amortised over what??

Sets in Haskell are trees themselves, and it is odd to store a tree in every node of a tree. Can you take the finger tree, specialized to this particular monoidal annotation, and optimize it further more?

5 Related Work

Approaches different to our own have, of course, been taken by other researchers, in particular by Dexter et al. [4] as well as by Headley and Hammer [7]. The former uses techniques to delay the computation of updates and the latter uses lighter and simpler data structures. Considering our own work alongside these approaches leads us to suggest that it may be possible to produce an interesting combination of the three, since there are at least three aspects we have found to be experimentally time-consuming in practice:

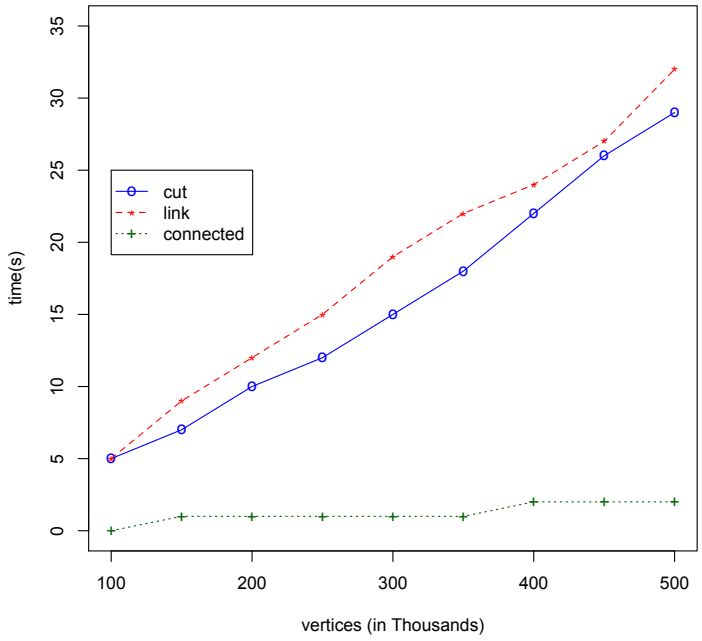

Fig. 10. Sequence of *link*s, *cut*s, and *connected*s

1. an Euler tour is treated as a sequence by efficient structures like finger trees, but the corresponding tree-like operations are also involved in update operations. A mixed, or lighter structure is an essential step towards achieving general efficiency;
2. the unbounded sequence of update operations results in a need to maintain the whole structure. In this context persistence triggers the consumption of memory allocation, and this suggests that benefits may accrue if we were to use higher order programming with effects;
3. the internal nodes of our finger tree structure hold the internal monoid `Set`, which is the most time-consuming element in our code and experimental results. Since rebalancing is an essential property of the internal binary search tree in `Set`, we propose conducting an analysis and design of different new balanced binary search tree as monoid in future work.

6 Conclusion and Future Work

Although we have presented evidence that ETFT trees are efficient data structures for dynamic tree handling, we have noticed experimentally that there is plenty of work to do specifically in the stream of interleaved operations applied as input to the data structure, that is, the unbounded sequence of updates and queries.

Finally, there is an evident need for a library of well-crafted test cases against which implementations of dynamic trees and graphs can be tested. At present, for example, there is little to guide us when generating the update-sequences against which our structures are validated, and this raises a number of obvious issues. For example,

- Can we find test sets which guarantee coverage of key properties?
- Which properties are we interested in?
- If we know the general ratio of updates to queries (say), can we identify a more specific data structure to enhance efficiency still further?
- Or maybe its enough to test solutions empirically, using live data from (say) the ever-changing structure of links in the Internet of Things?

These and other questions remain to be addressed, but we believe the quest for efficient dynamic data structures will become ever more important as seek to model, simplify and reason about the increasingly dynamic algorithmic structures in which modern culture is immersed and on which it depends.

Uniqueness on edges allow to carry labels, therefore ETFT could be the core for other approaches to dynamic trees such link-cut trees. Examples, illustrations, references ??

Acknowledgements: chahine.moreau@gmail.com ??

References

1. Acar, U.A., Blelloch, G.E., Harper, R., Vittes, J.L., Woo, S.L.M.: Dynamizing static algorithms, with applications to dynamic trees and history independence. In:

Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 531–540. Society for Industrial and Applied Mathematics (2004)
2. Blelloch, G., Ferizovic, D., Sun, Y.: Parallel ordered sets using join. arXiv preprint arXiv:1602.02120 (2016)
3. Demaine, E., Langerman, S., Price, E.: Confluently persistent tries for efficient version control. Algorithm Theory–SWAT 2008 pp. 160–172 (2008)
4. Dexter, P., Liu, Y.D., Chiu, K.: Lazy graph processing in haskell. In: Proceedings of the 9th International Symposium on Haskell. pp. 182–192. ACM (2016)
5. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM Journal on Computing 14(4), 781–798 (1985)
6. Gill, A.: Monoid. `https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Monoid.html` (2001), [Online; accessed 02-Dec-2017]
7. Headley, K., Hammer, M.A.: The random access zipper: Simple, purely-functional sequences. arXiv preprint arXiv:1608.06009 (2016)
8. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarithmic time per operation. Journal of the ACM (JACM) 46(4), 502–516 (1999)
9. Hinze, R., Paterson, R.: Finger tree. `http://hackage.haskell.org/package/fingertree` (2006), [Online; accessed 02-Dec-2017]
10. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of Functional Programming 16(02), 197–217 (2006)
11. Leijen, D.: Set. `https://hackage.haskell.org/package/containers-0.5.10.2/docs/Data-Set.html` (2002), [Online; accessed 02-Dec-2017]
12. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. Journal of computer and system sciences 26(3), 362–391 (1983)
13. Werneck, R.F.: Design and Analysis of Data Structures for Dynamic Trees. Ph.D. thesis, Princeton University (2006)
14. Yorgey, B.A.: Monoids: theme and variations (functional pearl). ACM SIGPLAN Notices 47(12), 105–116 (2012)