FunSeqSet: Towards a Purely Functional Data Structure for the Linearisation case of Dynamic Trees Problem

Juan C. Saenz-Carrasco

WFLP 2019
Outline

• Dynamic Trees Problem
• Applications
• Fundamentals
• FunSeqSet
• Experimental Results
• Further Work
Dynamic Trees Problem

• We refer to the term “dynamic trees problem” to the one defined by [Sleator and Tarjan,1983]:

“…maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: a link operation that combines two trees into one by adding an edge, and a cut operation that divides one tree into two by deleting an edge...”
Applications

• Among other, the following applications rely on the above link and cut operations:

  • Flows on Networks, [Tarjan 1983, Nanongkai et. al. 2019], for minimum or maximum cases

  • Rearrangement of Labelled Trees, [Bernardini et. al. 2019], for trees comparison in order to represent history in tumor cancer

  • Geomorphology, [Ophelders et. al., 2019], for modelling the dynamic behaviour of a riverbed.
Work done so far …

- There are at least three different representations for dynamic trees data structures:
  - Path Decomposition [Sleator and Tarjan, 1983]
  - Tree Contraction [Alstrup, 1997]
  - **Linearisation** [Henzinger and King, 1997; Tarjan 1997], also known as Euler-tour trees (ETTs)

- But … all of the above under the imperative programming setting
Fundamentals

- We deal with three kind of tree structures:

1. The $k$-degree tree, i.e. input tree, represented in Haskell by a rose tree

```haskell
data Tree a = Node { rootLabel :: a, subForest :: Forest a }

type Forest a = [Tree a]
```
Fundamentals

- Example of rose tree of small integers

Node 7 [Node 9 [ Node 5 []
  , Node 2 []
  , Node 4 []
  , Node 3 [] ]]

- Since there is no balancing scheme, query or updates operations on this tree take $O(n)$ time
2. The BST tree or the set, defined in Haskell in library Data.Set as

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

- Example of set of small integers

\[
\text{Bin 6 5 ( Bin 3 3 (Bin 1 2 Tip Tip) (Bin 1 4 Tip Tip) ) ( Bin 2 7 Tip (Bin 1 9 Tip Tip) )}
\]

- Since there is a balancing scheme, query and updates operations on this tree take $O(\log n)$ time each.
3. Finally, the finger tree (FT), for managing sequences (e.g. order of the elements is preserved). Devised by [Hinze and Paterson, 2006] is defined through three data types

```hs
data Node v a = Node2 v a a |
                Node3 v a a a
```

```hs
data Digit a = One       a |
                Two     a a |
                Three  a a a |
                Four    a a a a
```

data FingerTree v a
  = Empty
  | Single a
  | Deep v (Digit a) (FingerTree v (Node v a)) (Digit a)

- Example of a finger tree of integers where the monoidal annotation is an integer (i.e. Data.Sequence)
In order to deal with the sequence under a certain operations, a \textit{measurement} is introduce. For instance, looking for an element via its index. In the case of Data.Sequence

\begin{verbatim}
class Sized a where
  size :: a -> Int

instance Sized a => Sized (FingerTree a) where
  size Empty           = 0
  size (Single x)      = size x
  size (Deep v _ _ _)  = v
\end{verbatim}
## Performance in FTs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>view left (or right)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert from left $&lt;$ (or right $&gt;$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>appending $t_1$ $t_2$, $&gt;$</td>
<td>$O(\log \min t_1 t_2)$</td>
</tr>
<tr>
<td>searching</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Runtime is amortised, and depends of the performance of the monoidal annotation, since leaves of the FT do not perform any computation.
Euler-tour Tree as Sequence

Given a $k$-ary tree we flatten it by firstly defining an extra edge (one in each direction) to create a tour, and then select an arbitrary vertex (called the root) in order to create the “linear-shape”. We do not close the tour to avoid duplication for the root vertex.
Euler-tour Tree as Sequence

(7,7) → (7,9) → (9,9) → (9,5) → (5,5) → (5,9) → (9,2) → (2,2) → (2,9) → (9,4) → (4,4) → (4,9) → (9,3) → (3,3) → (3,9) → (9,7) → (9,3) → (3,9) → (9,7)

IV

(7,7) → (7,9) → (9,9) → (9,5)

1 → 2

II

(5,5) → (5,9) → (9,2) → (2,2)

5 → 3 → 4 → 2

N2

IV

(9,3) → (3,3) → (3,9) → (9,7)

→ 8 → 3 → 9 → 10

II

(2,9) → (9,4) → (4,4) → (4,9)

→ 5 → 6 → 4 → 7

N2
We follow [Tarjan, 1997]. “…Specifically, suppose \textit{link}\{(u, v)\} is selected. Let \(T_1\) and \(T_2\) be the trees containing \(u\) and \(v\) respectively, and let \(L_1\) and \(L_2\) be the lists representing \(T_1\) and \(T_2\). We split \(L_1\) just after \((u, u)\) into lists \(L_1^1, L_1^2\), and we split \(L_2\) just after \((v, v)\) into \(L_2^1, L_2^2\). Then, we form the list representing the combined tree by catenating the six lists \(L_1^2, L_1^1, [(u, v)], L_2^2, L_2^1, [(v, u)]\) in order…”
The above *declarative* procedure, can be implemented in Haskell as, but before the data types

```haskell
type Pair = (Int, Int)
type FT = FingerTree (S.Set Pair) Pair
type Forest = FingerTree (S.Set Pair) FT

instance Edges (S.Set Pair) Pair where
  edges (x,y) = S.insert (x,y) S.empty
```
link

\[
\text{linkTree} :: \text{Ord} \ a \Rightarrow a \to \text{FT} \ a \to a \to \text{FT} \ a \to \text{Maybe} \ (\text{FT} \ a)
\]
\[
\text{linkTree} \ u \ tu \ v \ tv = \text{Just} \$
\]
\[
\text{let} \ from = \text{rerooot} \ tu \ u \quad
\text{(Position} \ \text{left} _ \ _ \ \text{right}) = \text{search} \ \text{pred} \ tv \quad
\text{in} \quad ((\text{left} \ |\ (v,v)) \ |\ (v,u)) \ \text{<<} \ \text{from} \ \text{<<} ((u,v) \ \text{<<} \ \text{right})
\]
\[
\text{where} \quad
\text{pred} \ \text{before} _ = (\text{S}.\text{member} \ (v,v)) \ \text{before}
\]

rerooot :: \text{Ord} \ a \Rightarrow \text{FT} \ a \to a \to \text{FT} \ a
\[
rerooot \ \text{tree} \ \text{vertex} = \text{case} \ (\text{search} \ \text{pred} \ \text{tree}) \ of
\]
\[
\text{Position} \ \text{left} _ \ _ \ \text{right} \to \text{root} \ \text{<<} \ (\text{right} \ \text{<<} \ \text{left})
\]
\[
_ \to \text{tree}
\]
\[
\text{where} \quad \text{root} = (\text{vertex,vertex})
\]
\[
\text{pred} \ \text{before} _ = (\text{S}.\text{member} \ \text{root}) \ \text{before}
\]
link, safe?

\[
\text{linkTreeSafe} :: \text{Ord}\ a \to a \to \text{FT} \ a \to a \to \text{FT} \ a \to \text{Maybe} \ (\text{FT} \ a)
\]
\[
\text{linkTreeSafe} \ u \ tu \ v \ tv = \text{case} \ (\text{edgeIn} \ (u,u) \ tu, \ \text{edgeIn} \ (v,v) \ tv) \ \text{of}
\]
\[
\quad \text{(False, _)} \to \text{Nothing}
\]
\[
\quad (_\quad, \text{False}) \to \text{Nothing}
\]
\[
\quad \text{(True, True)} \to \text{Just} \$
\]
\[
\quad \text{let} \ \text{from} = \text{reroot} \ tu \ u
\]
\[
\quad \quad \text{(Position left _ right)} = \text{search} \ \text{pred} \ tv
\]
\[
\quad \text{in} \ ((\text{left } |> (v,v)) \ |> (v,u)) \gg \text{from} \gg ((u,v) <| \text{right})
\]
\[
\quad \text{where}
\]
\[
\quad \text{pred before _} = (S.\text{member} \ (v,v)) \text{ before}
\]

\[
\text{edgeIn} :: (\text{Measured} \ (S.\text{Set} \ a) \ b, \ \text{Ord} \ a) \Rightarrow a \to \text{FT} \ a \to \text{Bool}
\]
\[
\text{edgeIn} \ e \ ft = \text{case} \ (\text{search} \ \text{pred} \ ft) \ \text{of}
\]
\[
\quad \text{Position _ _ _} \to \text{True}
\]
\[
\quad (_\quad \quad \quad) \to \text{False}
\]
\[
\quad \text{where}
\]
\[
\quad \text{pred before _} = (S.\text{member} \ e) \text{ before}
\]
link (within a forest)

```haskell
data ForestFull a = ForestFull NumNodes ForestSize (FT a)
type NumNodes = Int -- of the form (v,v)
type ForestSize = Int -- Also is the total number of edges -- including NumNodes

link :: Ord a => a -> a -> ForestFull a -> ForestFull a
link x y forest@(ForestForest nnodes size ft)
  | x == y = forest
  | size > (2*nnodes) = linkDEN x y forest
  | otherwise = linkAVG x y forest
```
Searching for an element with NO index

(2,2), (2,9), (3,3), (3,9), (4,4), (4,9), (5,5), (5,9), (7,7), (7,9),(9,2), (9,3), (9,4), (9,5), (9,7), (9,9)

(7, 7)  (7, 9)  (9, 9)  (9, 5)

1  7  9  2

(2,2), (2,9), (4,4), (4,9), (5,5), (5,9), (9,2), (9,4)

(9, 2)  (5, 5), (5, 9)

3  5  4  2

(2,9), (9,4)

5  6  4  7
Performance in FTs with Set as monoidal annotation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Runtime bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>view left (or right)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert from left &lt;</td>
<td>(or right</td>
</tr>
<tr>
<td>appending t1 t2, &gt;&lt;</td>
<td>$O(\log (\text{min } t1 \ t2)) \times O(\text{union})$</td>
</tr>
<tr>
<td>searching</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

$O(\text{union}) = \log (m \times (\log (n/m) + 1))$, where $m \leq n$
complexity for the backbone in trees? in forest?

Before claiming a final complexity, let’s try to improve this structure a bit.
potential backbones (tree)

SEQ

- **No** Data.Set defined as monoid (Data.Sequence)
- “Light” allocation
- “Fast” insertion
- “Slow” searching as it traverses the whole structure
Potential backbones (tree)

\textbf{FULL}

- Data.Set defined as monoid everywhere (\textbf{full}) in the structure
- “Heavy” (fat?) allocation
- “Slow” insertion, as its stores sets in every internal node
- “Fast” searching
potential backbones (tree)

SEMI

- Data.Set defined as monoid only of the Node data constructor
- “semi” light allocation
- “semi” fast insertion
- Queries on (2 x height) of the structure as tree, and the whole structure when it is a forest
potential backbones (tree) TOP

• Data.Set defined as monoid only of the Node data constructor. Additional set is maintain outside the structure

• “medium” allocation

• “somewhat” insertion

• Queries on (2 x height) of the structure as tree and $O(\log n)$ as forest
potential backbones (forest)

SEQ

SEMI
potential backbones (forest)
potential backbones (forest)

Top

Finally, the Top-FT types

type Pair = (Int, Int)
type FT = FingerTree (S.Set Pair) Pair
data TopTree = TopTree { getSet :: S.Set Pair, getFT :: FT }
data TopForest = TopForest { getSetF :: S.Set Pair, getFTF :: Forest }

Plus the corresponding insertion, appending and search operations
Experimental Results, for backbones

All-FTs: Creating a forest of 50-node trees
Experimental Results, for backbones

Semi and Seq FTs: Querying a forest of 50-node trees
Experimental Results, for backbones

Full and Top FTs: Querying a forest of 50-node trees

- Full N/2
- Full N3/4
- Full N+1
- Full N/4
- Top N/2
- Top N3/4
- Top N+1
- Top N/4

Number of Operations (x10,000) vs. Time (µs)
Experimental Results, for backbones

All-FTs: Appending with Searching in a forest of 50-node trees
Experimental Results, for dynamic operations link, cut

Performance of link & cut operations in forests

- O(n) reference
- link & cut on 100-node forest
- link & cut on 500-node forest
- link & cut on 1000-node forest
Experimental Results, for dynamic operations link, cut

Time per individual operation link & cut interleaved

- **O(n) reference**
- **100-node Forest**
- **500-node Forest**
- **1000-node Forest**

![Graph showing time per individual operation link & cut interleaved for different forest sizes](image)
Further Work

- Specify the operations that belong to the Pre-Processing phase, such as generation of a forest.

- Analysis of potential parallel strategies in the pre-processing phase, for instance TOP-FT computes its top set separately from the insertion of elements into a tree or the tree into a forest.

- Define the theoretical complexity for the backbones (except SEQ, since it is done by [Hinze and Paterson, 2006]).

- Extend this work into Link-Cut trees approach by adding label annotation in the leaves of the FTs.
Source code and auxiliary data

• So far, Full FT is available as it the one referred by the paper at CoRR

https://github.com/jcsaenzcarrasco/FunSeqSet