Proving tree algorithms for succinct data structures

Reynald Affeldt [1] Jacques Garrigue [2]
Xuanrui Qi [3] Kazunari Tanaka [2]

[1] AIST

[2] Nagoya University

[3] Tufts University

November 22, 2018

https://github.com/affeldt-aist/succinct

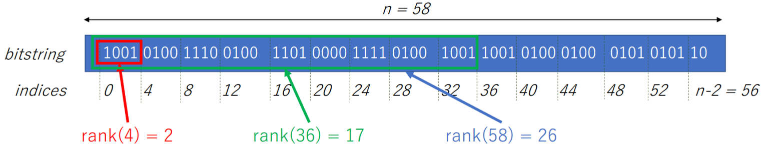
Succinct Data Structures

- Representation optimized for both time and space
- *“Compression without need to decompress”*
- Much used for Big Data
- Application examples
  - Compression for Data Mining
  - Google’s Japanese IME

Rank and Select

To allow fast access, two primitive functions are heavily optimized. They can be computed in constant time.

- rank(i) = number of 1's up to position *i*



n = *58*

bitstring 1001 0100 1110 0100 1101 0000 1111 0100 1001 1001 0100 0100 0101 0101 10

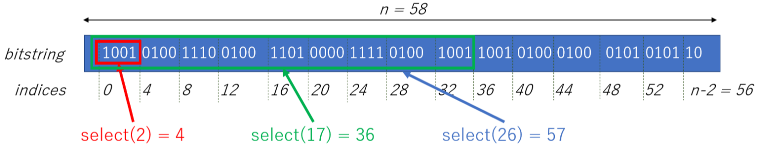
indices *0* *4* *8* *12* *16* *20* *24* *28* *32* *36* *40* *44* *48* *52* *n-2 = 56*

rank(4) = 2

rank(36) = 17

rank(58) = 26

- select(i) = position of the i^{th} 1: rank(select(i)) = i



n = *58*

bitstring 1001 0100 1110 0100 1101 0000 1111 0100 1001 1001 0100 0100 0101 0101 10

indices *0* *4* *8* *12* *16* *20* *24* *28* *32* *36* *40* *44* *48* *52* *n-2 = 56*

select(2) = 4

select(17) = 36

select(26) = 57

Proved implementation in [Tanaka A., Affeldt, Garrigue 2016]

Today's story

Trees in Succinct Data Structures

Featuring two views

As data Efficient encoding of trees using rank and select (this talk)

As tool Implementation of dynamic succinct data structures using red-black trees (next talk)

- Both are proved is Coq/SSReflect
- They can be combined together

Basic Coq definitions

rank is easily defined. select is its (minimal) inverse.

```
Variables (T : eqType) (b : T) (n : nat).
Definition rank i s := count_mem b (take i s).
Definition Rank (i : nat) (B : n.-tuple T) :=
  #|[set k : [1,n] | (k <= i) && (tacc B k == b)]|.
Lemma select_spec (i : nat) (B : n.-tuple T) :
  exists k, ((k <= n) && (Rank b k B == i)) ||
            (k == n.+1) && (count_mem b B < i).
Definition Select i (B : n.-tuple T) :=
  ex_minn (select_spec i B).
```

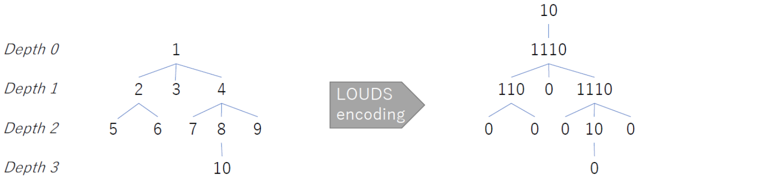
pred s y = last b up to y. succ s y = first b from y on.

```
Definition pred s y := select (rank y s) s.
Definition succ s y := select (rank y.-1 s).+1 s.
```

Getting the indexing right is a nightmare.
Here indices start from 1, but there is no fixed convention.

L.O.U.D.S.

Level-**O**rder **U**nary **D**egree **S**equence
[Navarro 2016, Chapter 8]



Depth 0	Depth 1	Depth 2	Depth 3
1	234	56789	10

	Depth 0	Depth 1	Depth 2	Depth 3
10	1110	11001110	000100	0

- Unary coding of node arities, put in breadth-first order
- Each node is arity 1's followed by a 0
- The structure of a tree uses just $2n + 2$ bits
- Useful for dictionaries (Google Japanese IME)

Implementation of primitives

We define an isomorphism between valid paths in the tree, and valid positions in the LOUDS.

The basic operations are

- Position of the root (2 with virtual root, counting from 0)
- Position of the i^{th} child of a node
- Position of its parent
- Number of children

```
Variable B : seq bool.
Definition LOUDS_child v i :=
  select false (rank true (v + i) B).+1 B.
Definition LOUDS_parent v :=
  pred false B (select true (rank false v B) B).
Definition LOUDS_children v :=
  succ false B v.+1 - v.+1.
```

First attempt

count_smaller t p = number of nodes appearing before the path p in breadth first order.

```
Definition LOUDS_position (t : tree) (p : seq nat) :=
  (count_smaller t p + (count_smaller t (rcons p 0)).-1).+2.
(*   number of 0's           number of 1's          virtual root *)

Definition LOUDS_subtree B (p : seq nat) :=
  foldl (LOUDS_child B) 2 p.

Theorem LOUDS_positionE t (p : seq nat) :
  let B := LOUDS t in valid_position t p ->
  LOUDS_position t p = LOUDS_subtree B p.

Theorem LOUDS_parentE t (p : seq nat) x :
  let B := LOUDS t in valid_position t (rcons p x) ->
  LOUDS_parent B (LOUDS_position t (rcons p x)) = LOUDS_position t p.

Theorem LOUDS_childrenE t (p : seq nat) :
  let B := LOUDS t in valid_position t p ->
  children t p = LOUDS_children B (LOUDS_position t p).
```

First attempt

Various problems

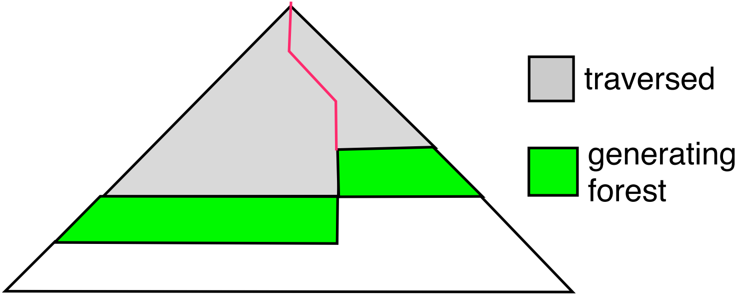
- Breadth first traversal does not follow the tree structure
- Cannot use structural induction
- No natural correspondence to use in proofs
- Oh, the indices!

As a result

- LOUDS related proofs take more than 800 lines
- Many lemmas have proofs longer than 50 line
- The should be a better approach...

Second try

- Introduce traversal up to a path
- For easy induction, work on forests rather than trees
- A generating forest need not be on the same level!



Traversal and Remainder

```
Variable (A B : Type) (f : tree A -> B).

(* Traversal of nodes before path p *)
Fixpoint lo_traversal_lt (w : forest A) (p : seq nat) : seq B.

(* Generating forest for nodes following path p *)
Fixpoint lo_traversal_res (w : forest A) (p : seq nat) : forest A.

(* Relation between them *)
Lemma lo_traversal_lt_cat w p1 p2 :
  lo_traversal_lt w (p1 ++ p2) =
  lo_traversal_lt w p1 ++ lo_traversal_lt (lo_traversal_res w p1) p2.

(* Complete traversals are all equal *)
Theorem lo_traversal_lt_max t p :
  size p >= height t ->
  lo_traversal_lt [:: t] p = lo_traversal_lt [:: t] (nseq (height t) 0).
```

All paths lead to Rome !

Indices and Positions in LOUDS

```
(* LOUDS_lt is a path-indexed traversal *)
Definition LOUDS_lt w p := flatten
  (lo_traversal_lt (node_description \o children_of_node) w p).

(* This corresponds to the standard definition of LOUDS *)
Theorem LOUDS_lt_ok (t : tree A) p :
  size p >= height t -> LOUDS t = true :: false :: LOUDS_lt [:: t] p.

(* Position of a node in the LOUDS *)
Definition LOUDS_position w p := size (LOUDS_lt w p).

(* Index of a node in level-order *)
Definition LOUDS_index w p := size (lo_traversal_lt id w p).

Lemma LOUDS_position_select w p p' :
  valid_position (head dummy w) p ->
  LOUDS_position w p =
  select false (LOUDS_index w p) (LOUDS_lt w (p ++ p')).

Lemma LOUDS_index_rank w p p' n :
  valid_position (head dummy w) (rcons p n) ->
  LOUDS_index w (rcons p n) =
  size w + rank true (LOUDS_position w p + n) (LOUDS_lt w (p ++ n :: p')).
```

Properties proved

```
Theorem LOUDS_childE (t : tree A) (p p' : seq nat) x :
  let B := LOUDS_lt [:: t] (rcons p x ++ p') in
  valid_position t (rcons p x) ->
  LOUDS_child B (LOUDS_position [:: t] p) x =
  LOUDS_position [:: t] (rcons p x).

Theorem LOUDS_parentE (t : tree A) p p' x :
  let B := LOUDS_lt [:: t] (rcons p x ++ p') in
  valid_position t (rcons p x) ->
  LOUDS_parent B (LOUDS_position [:: t] (rcons p x)) =
  LOUDS_position [:: t] p.

Theorem LOUDS_childrenE (t : tree A) (p p' : seq nat) :
  let B := LOUDS_lt [:: t] (rcons p 0 ++ p') in
  valid_position t p ->
  children t p = LOUDS_children B (LOUDS_position [:: t] p).
```

Bonus: A Structural Traversal

Breadth-first traversal uses induction on the height:

```
Variable f : tree A -> B.
Fixpoint lo_traversal'' n (l : forest A) :=
  if n is n'.+1 then
    map f l ++ lo_traversal'' f n' (children_of_forest l)
  else [::].
Definition lo_traversal t := lo_traversal'' (height t) [:: t].
```

We can avoid that by doing the traversal in 2 steps;
1st, build a list of levels, and then catenate them.

```
Fixpoint level_traversal t :=
  let: Node a cl := t in
  [:: f t] :: foldr (fun t1 => merge1 (level_traversal t1)) nil cl.

Fixpoint level_traversal_cat (t : tree A) ss {struct t} :=
  let: (s, ss) :=
    if ss is s :: ss then (s, ss) else (nil, nil) in
  let: Node a cl := t in
  (f t :: s) :: foldr level_traversal_cat ss cl.
Definition lo_traversal_cat t := flatten (level_traversal_cat t nil).
```

level_traversal is structural, but its complexity is bad.

Conclusion

Advantages of the new approach

- All proofs are by induction on paths
- Common lemmas arise naturally
- Down to about 500 lines in total, long proofs about 25

Remaining problems

- There are still long lemmas (`lo_traversal_lt_max`, . . .)
- Paths all over the place

Future work

- Can we apply that to other breadth-first traversals

Proofs

https://github.com/affeldt-aist/succinct