Proving tree algorithms for succinct data structures

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

[1]National Institute of Advanced Industrial Science and Technology, Japan

[2]Graduate School of Mathematics, Nagoya University, Japan

[3]Department of Computer Science, Tufts University, United States

April 24, 2019
Department of Computer Science, Boston University

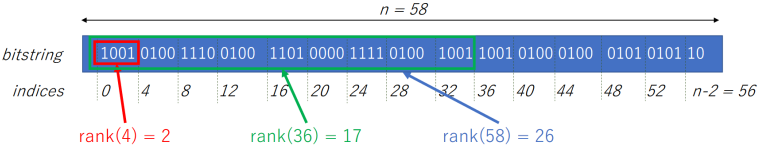
Succinct Data Structures

- Representation optimized for both time and space
- *"Compression without need to decompress"*
- Much used for Big Data
- Application examples
  - Compression for Data Mining
  - Mozc, the engine behind Google's Japanese IME

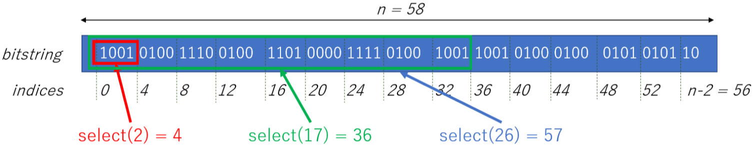
Rank and Select

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



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



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

As tool Implementation of dynamic succinct data structures using red-black trees

- Both are proved in 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).
```

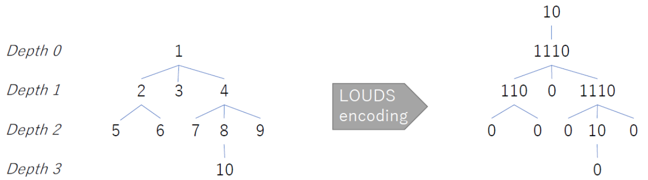
pred s y = last b up to y. succ s y = first b from y on.

```
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

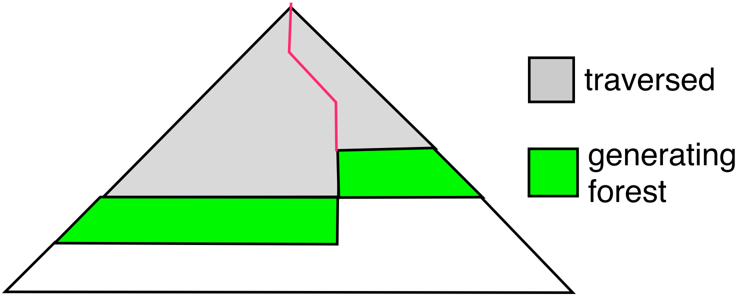
- Breadth first traversal does not follow the tree structure
- Cannot use structural induction
- No natural correspondence to use in proofs
- Oh, the indices!

As a result

- LOUDS related proofs take more than 800 lines
- Many lemmas have proofs longer than 50 lines
- 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).
```

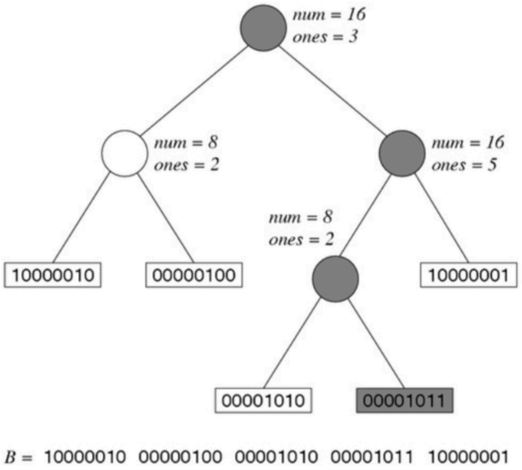
level_traversal is structural, but its complexity is bad.

Dynamic succinct data structures

- The most optimized representation of succinct data structures use arrays, and are not suitable for dynamic operations (insertions, etc.)
- However, we often want to modify our bit vectors.
- To minimize the cost of insertion and deletion, we must make a tradeoff: we no longer can access bits (or calculuate rank or select) in constant time.
- However, representing the vector as a balanced binary tree, we can implement all operations in $O(\log n)$ time.

[Navarro 2016, Chapter 12]

Representing dynamic bit vectors



num is the number of bits in the left branch and *ones* is the number of 1's in the left branch.

Our approach

- Red-black tree as our underlying data structure
  - The complexity results do not depend on the specific type of balanced binary tree used
  - Red-black trees are easy to implement purely functionally
  - Already multiple formalizations in Coq
  - However, due to the different data internals, we had to reimplement them
- We attempted two different approaches, using types in different ways:
  1. using ML types, i.e. only ML-style polymorphic ADTs
  2. using dependent types, all trees are correct by construction
- Proving the correctness of rank, insert and select

Implementation using ML types

Using red-black trees for bit vectors

```
Inductive color := Red | Black.
Inductive btree (D A : Type) : Type :=
| Bnode of color & btree D A & D & btree D A
| Bleaf of A.

Definition dtree := btree (nat * nat) (seq bool).
```

Assigning "meaning" to trees using `dflatten`

```
Fixpoint dflatten (B : dtree) :=
  match B with
  | Bnode _ l _ r => dflatten l ++ dflatten r
  | Bleaf s => s
  end.
```

Invariants of the internal data

```
Fixpoint wf_dtree (B : dtree) :=
  match B with
  | Bnode _ l (num, ones) r =>
    [&& num == size (dflatten l), ones == count_mem true (dflatten l),
        wf_dtree l & wf_dtree r]
  | Bleaf arr => (w ^ 2)./2 <= size arr < (w ^ 2).*2
  end.
```

Basic operations, using ML types

```
Fixpoint drank (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    if i < num then drank l i
                else ones + drank r (i - num)
  | Bleaf s =>
    rank true i s
  end.

Lemma dtree_ind (P : dtree -> Prop) :
  (forall c l r num ones,
   num = size (dflatten l) ->
   ones = count_mem true (dflatten l) ->
   wf_dtree l /\ wf_dtree r ->
   P l -> P r -> P (Bnode c l (num, ones) r)) ->
  (forall s, (w ^ 2)./2 <= size s < (w ^ 2).*2 -> P (Bleaf _ s)) ->
  forall B, wf_dtree B -> P B.

Lemma drankE (B : dtree) i :
  wf_dtree B -> drank B i = rank true i (dflatten B).
```

All those lemmas were proved in just a few lines.

Basic operations, using ML types

```
Fixpoint dselect_1 (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    if i <= ones then dselect_1 l i
                 else num + dselect_1 r (i - ones)
  | Bleaf s => select true i s
  end.

Fixpoint dselect_0 (B : dtree) (i : nat) :=
  match B with
  | Bnode _ l (num, ones) r =>
    let zeroes := num - ones in
    if i <= zeroes then dselect_0 l i
                   else num + dselect_0 r (i - zeroes)
  | Bleaf s => select false i s
  end.

Lemma dselect_1E B i :
  wf_dtree B -> dselect_1 B i = select true i (dflatten B).

Lemma dselect_0E B i :
 wf_dtree B -> dselect_0 B i = select false i (dflatten B).
```

Insertion, using ML types

```
Fixpoint dins (B : dtree) b i w : dtree :=
  match B with
  | Bleaf s =>
    let s' := insert1 s b i in
    if size s + 1 == 2 * (w ^ 2)
    then let n  := (size s') %/ 2 in
         let sl := take n s' in
         let sr := drop n s' in
         Bnode Red (Bleaf _ sl)
               (size sl, rank true (size sl) sl)
               (Bleaf _ sr)
    else Bleaf _ s'
  | Bnode c l (num, ones) r =>
    if i < num then balanceL c (dins l b i w) r
               else balanceR c l (dins r b (i - num) w)
  end.

Definition dinsert (B : dtree) b i w : dtree :=
  match dins B b i w with
  | Bleaf s => Bleaf _ s
  | Bnode _ l d r => Bnode Black l d r
  end.
```

Maintaining balance

- Much of the difficulties with red-black trees comes from the numerous cases arising in balancing operations.
- In the function `balanceL`, 11 goals were generated.
- We proved those using as little automation as possible, in idiomatic SSReflect style.

```
Ltac decompose_rewrite :=
  let H := fresh "H" in
  case/andP || (move=>H; rewrite ?H ?(eqP H)).

Lemma balanceL_wf c (l r : dtree) :
  wf_dtree l -> wf_dtree r -> wf_dtree (balanceL c l r).
Proof.
case: c => /= wfl wfr. by rewrite wfl wfr ?(dsizeE,donesE,eqxx).
case: l wfl =>
  [[[[] lll [lln llo] llr|llA] [ln lo] [[] lrl [lrn lro] lrr|lrA]
   |ll [ln lo] lr]|lA] /=;
  rewrite wfr; repeat decompose_rewrite;
  by rewrite ?(dsizeE,donesE,size_cat,count_cat,eqxx).
Qed.
```

Definitions, using dependent types

Invariants are guaranteed by the data constructors:

- as dynamic bit vectors
- as red-black trees

```
Definition is_black c := if c is Black then true else false.
Definition color_ok parent child :=
    is_black parent || is_black child.

Inductive tree : nat -> nat -> nat -> color -> Type :=
  | Leaf : forall (arr : seq bool),
      (w ^ 2)./ 2 <= size arr < (w ^ 2).*2 ->
      tree (size arr) (count_one arr) 0 Black
  | Node : forall {s1 o1 s2 o2 d cl cr c},
      color_ok c cl -> color_ok c cr ->
      tree s1 o1 d cl -> tree s2 o2 d cr ->
      tree (s1 + s2) (o1 + o2) (d + is_black c) c.
```

Operations, using dependent types

- The code and proofs for the basic queries are mostly unchanged
- No more `dtree_ind`!
- We defined `dins` using the `Program` environement

```
Program Fixpoint dinsert' {n m d c} (B : tree n m d c) (b : bool) i
     {measure (size_of_tree B)} : { B' : near_tree n.+1 (m + b) d c
                     | dflattenn B' = insert1 (dflatten B) b i } := ...
```

  20 obligations were generated, and were proved in 90 lines.

- Implementing `balanceL` と `balanceR`
  - At first, we could not seem to be able to define those using `Program`. Each was defined using 17 lines of tactics.
  - Later, we found out workarounds that enabled us to define them using `Program`. (`Program` is flaky.)

```
Definition balanceL {nl ml d cl cr nr mr} (p : color)
    (l : near_tree nl ml d cl) (r : tree nr mr d cr) :
    color_ok p (fix_color l) -> color_ok p cr ->
    {tr : near_tree (nl + nr) (ml + mr) (inc_black d p) p
    | dflattenn tr = dflattenn l ++ dflatten r}.
  destruct r as [s1 o1 s2 o2 s3 o3 d' x y z | s o d' c' cc r'].
  + case: p => //= cpl cpr.
  (* ... *)
```

The trouble with deletion

- Deletion from purely-functional red-black trees is known to be a difficult problem [Germane & Might 2014, Kahrs 2001]
- How to handle deletion gracefully while keeping all invariants intact?
- Take 1: directly transcribe [Kahrs 2001] to Coq
- Result: Our red-black tree deviates significantly from the standard version, making this transcription very hard

Dependent types to the rescue

Idea: use dependent types to guide our search for invariants, and transcribe the result to use ML types

- We code out the needed invariants for delete, and use dependent types to guide our search for a version of deletion that is right.
- First version: defined using 626 lines of Ltac (works, but ugly).
- Later rewritten using `Program`.

Back to deletion

- Insertion can be implemented just by adding a re-balancing operation.
- Deletion, on the other hand, require an intermediate structure.
  - dtree is not enough to represent unbalanced, intermediate states resulting from removing a bit.
  - We need to know whether the black height has decreased!
  - Solution: using a record to trace the extra information

```
Record deleted_dtree: Type := MkD { d_tree :> dtree;
  d_down: bool; d_del: nat * nat }.
Definition balanceL' (c: color) (l: deleted_dtree)
  (d: nat*nat) (r: dtree): deleted_dtree.
Definition balanceR' (c: color) (l: dtree)
  (d: nat*nat) (r: deleted_dtree): deleted_dtree.
```

- Also, we handled “bit borrowing”.

Proving programs with SSReflect

- Small-scale reflection is mainly used for mathematical proofs via MathComp
- However, recently becoming more popular for proofs of programs as well [Sergey, Sergey & Nanevski 2015]
- Good built-in library support for doing mathematics, but less so for inductive data

First impressions

- Small-scale reflection uses boolean predicates (instead of inductive propositions) to model program properties
- Our experience with boolean predicates is very positive: simplify reasoning, “built-in” automation

First impressions

- Small-scale reflection uses boolean predicates (instead of inductive propositions) to model program properties
- Our experience with boolean predicates is very positive: simplify reasoning, “built-in” automation
- One downside: they don’t decompose naturally in inductive proofs

First impressions

- Small-scale reflection uses boolean predicates (instead of inductive propositions) to model program properties
- Our experience with boolean predicates is very positive: simplify reasoning, “built-in” automation
- One downside: they don’t decompose naturally in inductive proofs
- Solution: alternative inductive principle (`dtree_ind`)

Automation for SSReflect

- Basic approach: case analysis, then "crush" (à la [Chlipala 2013]) all goals
- We wrote our own SSReflect-style tactic for crushing boolean predicate goals:

Automation for SSReflect

- Basic approach: case analysis, then "crush" (à la [Chlipala 2013]) all goals
- We wrote our own SSReflect-style tactic for crushing boolean predicate goals:

```
Ltac decompose_rewrite :=
  let H := fresh "H" in
  move/andP => [] || (move => H; try rewrite H; try rewrite (eqP H)).
```

Automation for SSReflect

- Basic approach: case analysis, then "crush" (à la [Chlipala 2013]) all goals
- We wrote our own SSReflect-style tactic for crushing boolean predicate goals:

```
Ltac decompose_rewrite :=
  let H := fresh "H" in
  move/andP => [] || (move => H; try rewrite H; try rewrite (eqP H)).
```

- Result: worked very well in general; only one lemma seems to not be handled well by this approach.
- Also, SSReflect lacked arithmetic automation, which turned out to be very useful.

LOUDS formalization

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

Dynamic bit-vector formalization

- Summary
  - SSReflect helped immensely, and we were able to write proofs in an idiomatic SSReflect style.
  - Our proof of red-black tree balance was especially clear and concise (compare [Appel 2011] and [Chlipala 2013]).
  - However, there were a lot of small lemmas.
  - Crushing approach worked very well as always
  - Dependent types: good experiment resulting in clean lemma environment, but dirty code

- Future work
  - Can we improve the dependently typed version?
  - Proofs about complexity
    - Time complexity: do we need to amortize? (Amortization can be tricky)
    - Space complexity: no previous work on this; how to define the corrrect predicates?

Dynamic bit-vector formalization

- Summary
  - SSReflect helped immensely, and we were able to write proofs in an idiomatic SSReflect style.
  - Our proof of red-black tree balance was especially clear and concise (compare [Appel 2011] and [Chlipala 2013]).
  - However, there were a lot of small lemmas.
  - Crushing approach worked very well as always
  - Dependent types: good experiment resulting in clean lemma environment, but dirty code

- Future work
  - Can we improve the dependently typed version?
  - Proofs about complexity
    - Time complexity: do we need to amortize? (Amortization can be tricky)
    - Space complexity: no previous work on this; how to define the corrrect predicates?

Proofs

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

Our paper

Preprint available on the arXiv (CoRR).

Our paper

Preprint available on the arXiv (CoRR).

Reynald Affeldt, Jacques Garrigue, Xuanrui (Ray) Qi, Kazunari Tanaka. Proving Tree Algorithms for Succinct Data Structures. arXiv:1904.02809 [cs.PL].

https://arxiv.org/abs/1904.02809

More in the paper

- Details of the algorithms we formalized
- Important lemmas in our proof development
- More discussion on the benefits/drawbacks to using SSReflect