Proving tree algorithms for succinct data structures

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

[1] 産業技術総合研究所

[2] 名古屋大学多元数理科学研究科

[3] Tufts University

August 30, 2018

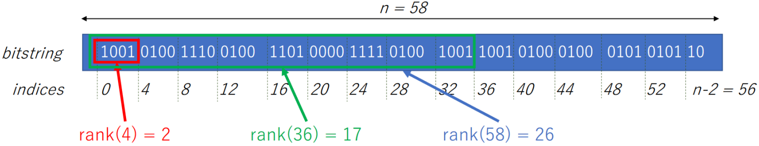
簡潔データ構造

- 時間・空間複雑さが共に良いデータ表現
- 「復号化の要らない圧縮」
- ビッグ・データでは多用されている
- 応用例
  - データマイニングのデータの圧縮
  - グーグル日本語 IME の辞書

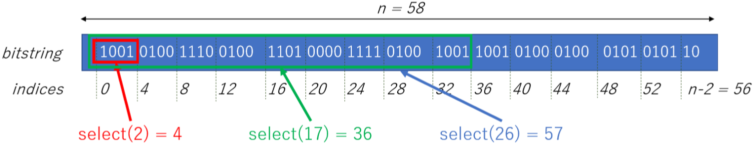
Rank と Select

ビット列への高速のアクセスを実現するために，二つの基本的な関数を最適化する．定数時間で実装可能．

- rank(i) ＝ i ビット目までの 1 の数



- select(i) ＝ i 番目の 1 の位置: rank(select(i)) ＝ i



[Tanaka A., Affeldt, Garrigue 2016] で実装を Coq で証明

今日の話

簡潔データ構造における木構造の表現と利用

二つの視点

表現 rank と select を使って, 木構造をビット列で表現する

利用 木構造 (red-black tree) を使って, 動的変化が可能なビット列を実装

- 両実装の基本性質を Coq/SSReflect で証明
- 前者を後者の上で利用できる

Coq での基本定義

rank は簡単に定義できる. select はその逆関数.

```
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 は y 以前の最後の b. succ s y は y 以後の最初の b.

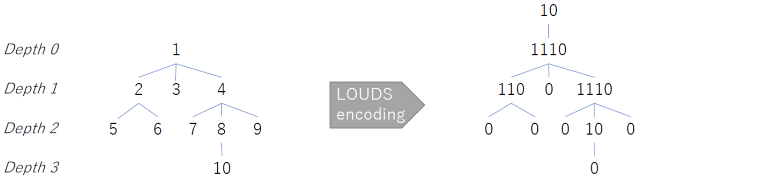
```
Definition pred s y := select (rank y s) s.
Definition succ s y := select (rank y.-1 s).+1 s.
```

添字を正しく合わせるのが大変.
ここでは添字を１から数えるが、本によって異なる.

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

- 幅優先に並べた各ノードの次数の一進数表記
- 各ノードの子の数を表す 1 の後に 0 を付ける
- 丁度長さ $2n + 2$ のビット列で木の分岐構造が書ける
- 辞書などに応用 (グーグル日本語 IME)

基本操作の実装

木の中のパスと LOUDS 列の中の位置の間に同型を定義する.

必要な操作は

- 根の位置 (疑似ルートの追加で 2, 位置は 0 から数える)
- i 番目の子ノードの位置
- 親ノードの位置
- 子供の数

```
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.
```

基本操作の実装

木の中のパスと LOUDS 列の中の位置の間に同型を定義する.

必要な操作は

- 根の位置 (疑似ルートの追加で 2, 位置は 0 から数える)
- i 番目の子ノードの位置
- 親ノードの位置
- 子供の数

```
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.
```

問題: 構造的な対応になっていない

具体的な対応

幅優先操作においてパスpより前に現れるノードの数をcount_smaller t pとする

```
Definition LOUDS_position (t : tree) (p : seq nat) :=
  (count_smaller t p + (count_smaller t (rcons p 0)).-1).+2.
(*       0 の数                 1 の数                 疑似ルート *)

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).
```

L.O.U.D.S の難しさ

様々な問題点

- 幅優先走査が木の構造から離れている
- 構造的な帰納法が使えない
- 欲しい性質が簡単に証明できる「自然な対応」がない
- 証明すると添字が中々合わない

結果的には

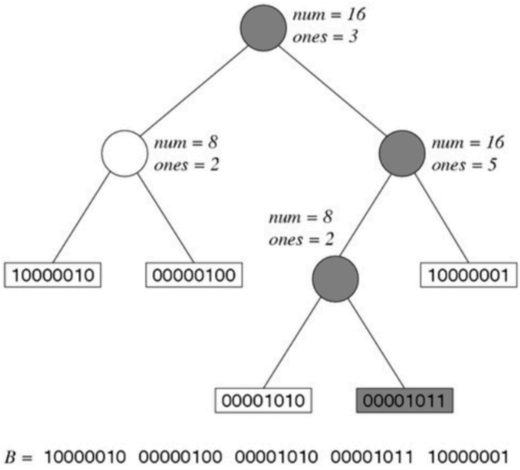
- LOUDS に関する証明は 800 行以上
- 50 行を越える補題を複数含む
- もっと良いアプローチをまだ模索中

動的簡潔データ構造

- 簡潔データ構造の最適な表現は配列を使うので，変更に不向きである
- しかし，データを動的に変えたい場合が多々ある
- 挿入・削除のコストを抑えるために，アクセスや rank・select の定数時間を諦めなければならない
- 表現に平衡木を使うと，全ての操作が $O(\log n)$ で行える

[Navarro 2016, Chapter 12]

動的ビット列の表現



num は左部分木のビット数, *ones* は左部分木の 1 の数

実装方針

- 平衡木として red-black tree を使用
  - 複雑さの結果は平衡木の種類に寄らない
  - 純粋な関数型プログラミング言語で表現しやすい
  - 既に Coq で複数の形式化がある
  - ただし，データの配置が異なるので再実装した
- 型の使い方の異なる 2 つの実装を試みた
  1. 「通常」の型を使った一階述語による実装
  2. 依存型を使い，正しいデータしか書けない実装
- rank・select・insert の実装の証明

通常型による実装

red-black tree をビット列に応用

```
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).
```

dflattenで木の意味を定義する

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

内部データが守るべき不変量

```
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 _ => true
  end.
```

通常型での基本操作

```
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, 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).
```

どちらも証明が数行で収まる

通常型での基本操作

```
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).
```

通常型での挿入

```
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.
```

平衡を保つ

- 平衡化の場合の多さが red-black tree の証明の難点
- `balanceL` に対する場合わけが 11 個のゴールを生成する
- SSReflect らしく，最低限の自動化で対応する

```
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.
```

依存型による定義

データ構造で不変量を保証する

- 動的ビット列として
- red-black tree として

```
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.
```

依存型での操作

- 基本操作は定義も証明もほとんど変わらず
- dtree_indは要らない
- dinsは Program 環境で定義できた

```
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 個の Obligation が生成され，全ての性質の証明は 90 行

- balanceL とbalanceRの定義
  - 回避できないバグにより Program 環境が使えなかった
  - 17 行で tactic による定義を完成させた

```
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.
  (* さらに 11 行で定義と証明が完成 *)
Defined.
```

動的ビット列のまとめと課題

- 通常型による証明
  - うまく SSReflect が利用でき，すっきりした証明
  - 特に，平衡化の証明の場合分けが従来研究より直感的
  - ただ，細々とした補題が多い
- 依存型による証明
  - 欲しい性質が型で表現され，証明の細分化を防ぐ
  - Program 環境のバグにより，読みにくい定義になる
  - 証明の修正が難しい
- 今後の課題
  - 依存型で削除も定義・証明できたが，改良の余地あり
  - 複雑さに関する証明も行いたい
    妥当な複雑さの定義が必要

動的ビット列のまとめと課題

- 通常型による証明
  - うまく SSReflect が利用でき，すっきりした証明
  - 特に，平衡化の証明の場合分けが従来研究より直感的
  - ただ，細々とした補題が多い
- 依存型による証明
  - 欲しい性質が型で表現され，証明の細分化を防ぐ
  - Program 環境のバグにより，読みにくい定義になる
  - 証明の修正が難しい
- 今後の課題
  - 依存型で削除も定義・証明できたが，改良の余地あり
  - 複雑さに関する証明も行いたい
    妥当な複雑さの定義が必要

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