二分木のランク計算のクリーン可逆シミュレーション

2014SE081 大久保雄飛　2014SE112 安田拓也

指導教員：横山哲郎

1　はじめに

本研究はランク計算・アンランク計算アルゴリズムの基に行われている．このアルゴリズムは Knott の 1977 年の論文 [3] において初めて作られた．Knott による 2 つのアルゴリズムはともに時間計算量が $O(n^2)$ である．1985 年には Er によって時間計算量が $O(n \log n)$ であるが，予め計算テーブルが用意されている場合は $O(n)$ であるようなランク計算アルゴリズムが作られた [5]．その他に 1977 年に Ruskey たちによって計算量が $O(n \log n)$ であるアルゴリズムが作られている [4]．これらのアルゴリズムは二分木に代表される木構造の列挙を行うアルゴリズムやランダム生成プログラムの基礎として知られており，重要なアルゴリズムの一つである．しかし，現在筆者の知る限り，Knott によるランク・アンランク計算アルゴリズムのみが可逆化されており [2]，まだその他のランク・アンランク計算アルゴリズムの可逆化は行われていない．また，一般的に，(非可逆な) アルゴリズムをクリーン可逆化する方法として Bennett 法 [6] が存在するが，時間計算量・空間計算量が悪化することが分かっており，元のアルゴリズムの時間計算量・空間計算量と漸近的に等しく，ゴミ出力が最適である可逆シミュレーションを生成する一般解法は現在わかっていない．よって，効率的なクリーン可逆シミュレーションは，手動で作るしかないのが現状である．本研究では，ランク・アンランク計算アルゴリズムのうち，Er によるランク・アンランク計算を行う Pascal プロシージャを基に，Er のランク・アンランク計算アルゴリズムのクリーン可逆化を行う．ランク・アンランク計算アルゴリズムは単射であることから必ずゴミ出力を無くすことができるため，クリーン可逆化を行うアルゴリズムとしては平易である．Er のランク・アンランク計算アルゴリズムのクリーン可逆化によってランク・アンランク計算アルゴリズムの族についての，可逆シミュレーションの導出に関する知見を得られることが期待される．

2　準備

この章では，本研究で用いたプログラミング言語 Janus についてや，ランク計算を行う上で必要な二分木の表現方法，埋め込み，可逆シミュレーションについて説明する．

2.1　Janus

可逆プログラミング言語 Janus[1] は，C 言語に似た構文に加えて可逆性を保証するための構文要素をもつ．変数の基本型は，整数型，整数型の配列とスタックである．今回用いる Janus は単射関数しか表すことができないという意味で可逆であることが証明されている．

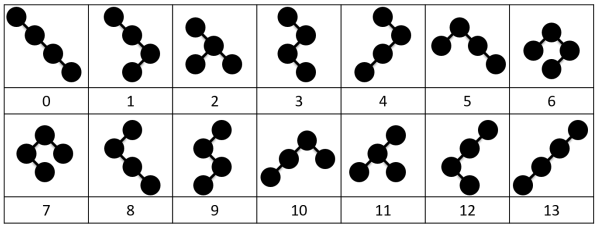

図 1　大きさ 4 の二分木とランク

2.2　二分木の表現

Er によるランク計算アルゴリズムにおいて利用されている二分木の表現方法である整形ビットパターンについて説明する．

節点数が n の二分木 T は $n+1$ 個の葉を追加することで拡張二分木 $\hat{T}$ に変換することができる．また，二分木は 0 または 1 のビットパターンに符号化することができ，葉は 0，節点は 1 を表すとして，根から前順序順に走査することで符号化できる．節点数が n である二分木 T に対応する拡張二分木 $\hat{T}$ は，$2n+1$ 個のビットからなるビットパターンに符号化される．このとき，生成されたビットパターンの最後のビットは必ず 0 になるため，最後のビットを除いた $2n$ 個のビットで 1 と 0 を用いて表現可能である．今，節点数が n の拡張二分木 $\hat{T}$ に対応する $2n$ 個のビットからなるビットパターンを $B = (b_1, b_2, ..., b_{2n})$ と表す．ここで b_i は，0 または 1 のビットである．ビットパターン B は，B を b_1 から b_{2n} まで走査したどの過程においても，1 のビットの総数と 0 のビットの総数を下回らず，1 のビットの総数と 0 のビットの総数が等しいとき，B のことを整形ビットパターンと呼ぶ．

2.3　ランク計算

二分木のランクの定義とその計算方法について述べた後，実際に例を用いて二分木のランク決めを行う．本稿でのランクとは同じ節の数である二分木に対して大小の大きさを決めることである．図 1 は節点数が 4 の二分木をランク順にならべた表である．Er による論文 [5] によると以下の式でランクを計算することができる．

$$Rank(B_n) = \sum_{b_i=1} V(i) \tag{1}$$

ただし，ここで $V(i)$ は b_{i+1} から b_{2n} の間で，$f(i)$ 個の 1 のビットと $(2n-i-f(i))$ 個の 0 のビットからなる整形ビットパターンの総数のことであり，$V(i)$ は次の二項係数で表される式で求められる．

$$V(i) = \binom{2n-i}{f(i)} - \binom{2n-i}{f(i)-1} \tag{2}$$

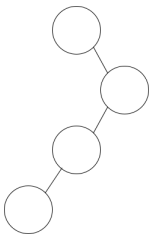

図 2　ランクが 4 の二分木

ここで $f(i)$ は整形ビットパターンの b_i から b_{2n-1} の間の 1 のビットの総数のことである．

実際に例を用いてランク計算を行う．図 2 は節点数 4 の二分木である．まずこの二分木を 8 個のビットからなるビットパターンに符号化すると，10111000 と表すことができる．次に，式 (1) を利用すると

$$\begin{aligned} Rank(10111000) = &\binom{2\times4-1}{4} - \binom{2\times4-1}{3} \\ &+ \binom{2\times4-3}{3} - \binom{2\times4-3}{2} \\ &+ \binom{2\times4-4}{2} - \binom{2\times4-4}{1} \\ &+ \binom{2\times4-5}{1} - \binom{2\times4-5}{0} \end{aligned}$$

となり，これを計算するとランクが 4 と求めることができる．

2.4　埋め込み

ここでは埋め込みについて定義を述べた後で，実際に例を用いて説明する．埋め込みとは非可逆計算で失われる情報を全て記憶する変換のことである．非可逆計算で失われる情報には，代入を行う時の代入前の値や，条件分岐の時，分岐後の値がどちらの分岐から得られたのかを特定する制御情報などがあげられる．`x=e;` のような代入文は，代入前の `x` の値を特定することができないので非可逆である．この文に埋め込みを適用すると `push(x,g); x+=e;` と書き換えることができる．`g` はゴミスタックである．代入前の `x` の値をゴミスタック `g` に保存することで代入前の値も求めることができる．　図 3 は条件分岐の概念図とプログラムをあらわしている．図 3 の左は非可逆なものであり，図 3 の右は埋め込みを適用したものである．条件分岐では `then` 節から得られたものなのか `else` 節から得られたものなのかを非可逆プログラムでは特定できないが，埋め込みを用いるとゴミスタックに情報を保存しているので，`then` 節から得られたのか `else` 節から得られたのかを特定することができる．

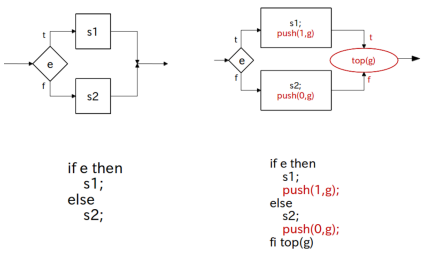

図 3　条件分岐の埋め込み

2.5　可逆シミュレーション

ここでは可逆シミュレーションやクリーン可逆シミュレーションなどを図を用いて説明する．図 4 の p はアルゴリズムであり，q は可逆アルゴリズムである．図 4 の四角い部分の左側は入力をあらわしており，右側は出力をあらわしている．出力のうち G はゴミ情報をあらわしている．可逆アルゴリズム q を p の可逆シミュレーションと呼ぶ．同じ可逆アルゴリズムでも出力にゴミ情報があらわれないものをクリーンと呼び，図 4 の右図は q のクリーン可逆シミュレーションである．

3　ランク計算の可逆シミュレーションの実現

Er[5] によるランク・アンランク計算プロシージャの可逆化を行う．まず，ランク・アンランク計算プロシージャそれぞれの埋め込みによる可逆シミュレーションを実現する．次に，それらの可逆シミュレーションの組み合わせと，一般解法の一つである Bennett 法 [6] を用いて複数パスのクリーン可逆シミュレーションを実現する．最後に，我々は 1 パスで空間計算量が効率の良いクリーン可逆シミュレーションを提案する．

3.1　埋め込みによるランク計算

図 5 のプロシージャ `rankE` は Er によるランク計算プロシージャを，埋め込みを用いて可逆化したプロシージャである．引数のスタック `g` はゴミ出力を表している．スタック `g` には出力に必要なくなった整形ビットパターンが埋め込まれており，図 5 において，27 行目の `clear_array` というプロシージャの呼出しで引数 `B` の各要素がゴミスタック `g` に格納されている．局所変数 `cnt` は `B[i:2*n-1]` に含まれる 1 の個数であり，式 2 中の $f(i)$ に対応する．17 行目で `B[i]` が 1 である場合のみ，18 行目と 20 行目で $V(i)$ を計算し，19 行目で `cnt` を 1 減らしている．`rankE` は，埋め込みだけでなく，可逆性を満たすために，元の Pascal プロシージャには無かった `if` 文実行後のアサーション `fi` を追加している．また，非可逆的な代入文は，19 行目のように，可逆な複合代入演算子の `+=`や`-=`で同じ処理になるように置き換えている．この埋め込みによるランク計算プロシージャは，入力の一つである整形ビットパターン `B` をゼロクリアするために，27 行目でゴミスタック `g` に埋め込んでいるため，入力された整形ビットパター

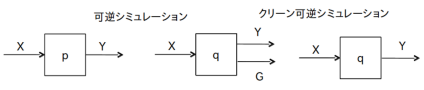

図 4　可逆シミュレーションの概念図

```
1 procedure clear_array (int a[], int n, stack g)
2     local int i = 0
3     from i = 0 do
4         local int t = a[i]
5         a[i] ^= t
6         push (t, g)
7         delocal int t = 0
8         i += 1
9     until i = n
10    delocal int i = n
11
12 procedure rankE(int p, int B[], int n, stack g)
13    local int cnt = n
14    local int i = 0
15    from i = 0 do
16        local int posn = 2*n-i-1
17        if B[i] = 1 then
18            call binom(posn, cnt, p)
19            cnt -= 1
20            uncall binom(posn, cnt, p)
21        fi B[i] = 1
22        delocal int posn = 2*n-i-1
23        i += 1
24    until i = 2*n-1
25    delocal int i = 2*n-1
26    delocal int cnt = 0
27    call clear_array(B, 2*n, g)
```

図 5 埋め込みによるランク計算

ンの長さ $2n$ に比例したメモリを使用してしまうため，空間計算量が元のプロシージャよりも悪化している．この埋め込みによるランク計算プロシージャは，制御の流れが同じため，時間計算量は元のプロシージャの時間計算量 T とおいた場合，$\mathrm{O}(T)$ となる．

次にこの埋め込みによるランク計算が元の Pascal プロシージャの可逆シミュレーションになっていることを示す．節数 n の二分木 T に対応するビット数 $2n$ の整形ビットパターン B について，

$$[\![\texttt{rankE}]\!]^{\texttt{Janus}} : (B, n) \mapsto ((p, n), g) \tag{3}$$

となる．式 3 において，p は二分木 T のランクである．また，0 や nil となる入力と出力は省略した．ここで，任意の B と n に対して

$$fst([\![\texttt{rankE}]\!]^{\texttt{Janus}}(B, n)) = [\![\texttt{rankP}]\!]^{\texttt{Pascal}}(B, n+1) \tag{4}$$

であるので，Janus プログラム rankE は Pascal プログラム rankP の可逆シミュレーションになっている．

3.2 埋め込みによるアンランク計算

図 6 のプロシージャ rankE は Er によるアンランク計算プロシージャを，埋め込みを用いて可逆化したプロシージャである．この可逆プロシージャ unrankE の時間計算量は，元のプロシージャと制御構造が同じであるため，時間計算量は元のプロシージャの時間計算量 T に対して，$\mathrm{O}(T)$ となる．unrankE の空間計算量は，14 行目でのゴミスタックへの格納が $2n-1$ 回繰り返されるため，$\mathrm{O}(n)$ となる．そのため，元のプロシージャが一定数の変数しか使用していないことに比べてメモリ使用量が悪化している．埋め込みによるアンランク計算が元の Pascal プロシージャの可逆シミュレーションになっていることは，埋め込みによるランク計算計算が元の Pascal プロシージャ

```
1 procedure unrankE(int p, int B[], int n, stack g)
2     local int cnt = n
3     local int i = 0
4     from i = 0 do
5         local int posn = 2*n-i-1
6         local int v = 0
7         call binom(posn, cnt, v)
8         uncall binom(posn, cnt-1, v)
9         if p >= v then
10            B[i] ^= 1
11            p -= v
12            cnt -= 1
13        fi B[i] = 1
14        push(v,g)
15        delocal int v = 0
16        delocal int posn = 2*n-i-1
17        i += 1
18    until i = 2*n-1
19    delocal int i = 2*n-1
20    delocal int cnt = 0
```

図 6 埋め込みによるアンランク計算

の可逆シミュレーションになっていることと同様に示すことができる．

3.3 Bennett 法によるクリーン可逆シミュレーション

前節におけるランク計算とアンランク計算は，元のプロシージャの可逆シミュレーションとなっているが，どちらの可逆シミュレーションも出力に元の出力にはなかったゴミスタックが含まれており，クリーンではない．Bennett 法を用いることで一般のアルゴリズムについてクリーン可逆シミュレーションを作ることができ，論文 [2] での，Bennett 法による Knott のランク・アンランク計算アルゴリズムのクリーン可逆シミュレーションと同様に作ることができる．Bennett 法によるランク計算のクリーン可逆シミュレーションでは，rankE および unrankE プロシージャの呼出しと逆呼び出しが合計 4 回行われているため，実行時間について効率が悪化していることがわかる．また，クリーンであるため最終的な出力としてゴミを出力しないが，rankE 及び unrankE の呼出しで中間的にゴミを生成しており，メモリ使用量が多くなってしまっている．

3.4 ランク計算のクリーン可逆シミュレーション

前節の Bennett 法によるランク計算のクリーン可逆シミュレーションは，時間計算量と空間計算量の両方の効率が悪くなっていた．図 7 のプロシージャ unrank は，節 3.1 と節 3.2 で示した可逆プロシージャ rankE 及び unrankE を組み合わせて作成した可逆プロシージャである．

unrankE の 7 行目及び 8 行目で計算されている数と，rankE の 18 行目から unrankE の 14 行目で v がゴミスタック g に格納されているが，lst:call2 行目で計算されている数が一致することより，rankE の 7 行目及び 8 行目の逆計算を行うことで，unrankE の 14 行目で v をゴミスタック g に格納する必要がなくなる．また，rankE の 27 行目の B のゼロクリアは，unrankE の 10 行目の B[i] に 1 を可逆代入する文の逆計算で置き換えることができる．プロシージャ unrank 自体はアンランク計算プロシージャであるが，逆呼出しすることで，ランク計算が可能である．

```
1 procedure unrank(int p, int B[], int n)
2     local int cnt = n
3     local int i = 0
4     from i = 0 do
5         local int posn = 2*n-i-1
6         local int v = 0
7         call binom(posn, cnt, v)
8         uncall binom(posn, cnt-1, v)
9         if p >= v then
10            B[i] ^= 1
11            p -= v
12            uncall binom(posn, cnt, v)
13            cnt -= 1
14            call binom(posn, cnt, v)
15        else
16            uncall binom(posn, cnt, v)
17            call binom(posn, cnt-1, v)
18        fi B[i] = 1
19        delocal int v = 0
20        delocal int posn = 2*n-i-1
21        i += 1
22    until i = 2*n-1
23    delocal int i = 2*n-1
24    delocal int cnt = 0
```

図 7 クリーン可逆シミュレーション

3.5 提案するクリーン可逆シミュレーションの計算量

今回提案する図 7 のランク計算のクリーン可逆シミュレーションの時間計算量は，$O(n \log n)$ となる．次に $O(n \log n)$ となることを示す．Shamir[8] による論文によれば，整数 n，k に対応する二項係数 $\binom{n}{k}$ の時間計算量は，$O(\log n)$ となることが分かっている．プロシージャ `unrank` の一回のループにおいて，二項係数の計算は 4 回行われており，$2n-1$ 回反復するので，合計計算ステップは $4(2n-1)O(\log n)$ となる．よって時間計算量は $O(n \log n)$ である．可逆シミュレーションは，ゴミ出力が有界で，元の非可逆なプロシージャと同じかそれ以下の時間計算量であり，追加で必要となるメモリの大きさが入力 x のメモリ上で表現する際の大きさ $|x|$ に対して高々 $g(|x|)$ であるとき，ゴミが g によって抑えられる忠実的な可逆シミュレーションと呼ぶ．さらに，ゴミが g で抑えられる忠実的な可逆シミュレーションよりもゴミ出力が小さい忠実的な可逆シミュレーションが存在しない場合，衛生的な可逆シミュレーションと呼ぶ [7]．プロシージャ `unrank` はゴミ出力が無く，元の Pascal プロシージャと時間計算量が同じであり，メモリを追加で必要としないため，衛生的な可逆シミュレーションである．

4 おわりに

本研究では，Er によるランク・アンランク計算アルゴリズムの効率的なクリーン可逆シミュレーションの実現を行った．作成したクリーン可逆シミュレーションは，一般解法によって作られたクリーン可逆シミュレーションと比較して，時間計算量及び空間計算量が漸近的に同じであるが，4 パスであった実行パスが 1 パスになっており，中間ゴミが作られないため，元の非可逆アルゴリズムと漸近的に同じ計算効率のクリーン可逆シミュレーションを作ることができた．

先行研究として Knott によるランク・アンランク計算アルゴリズムの同様な研究がある．この二つのランク・アンランク計算アルゴリズムを比較すると，可逆なプログラムの上での二分木の表現が異なっており，Knott のアルゴリズムでは，木置換で表現していたが，今回の Er のアルゴリズムでは，整形ビットパターンで表現していた．また，二分木の順序の与え方についてみると，Knott のアルゴリズムでは，自然順序が使われていたのに対して，Er のアルゴリズムでは辞書順が使われていたため，二分木の並び順が異なっている．

今後の課題は，その他のランク・アンランク計算アルゴリズムの効率的なクリーン可逆シミュレーションを提案することと，今回クリーン可逆化を行った Er によるランク・アンランク計算アルゴリズム及び Knott によるアルゴリズムとの関連性を探ることが挙げられる．

参考文献

[1] Yokoyama, T., Glück, R.: A Reversible Programming Language and its Invertible Self-Interpreter, Proc. *the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp.144–153 (2007).

[2] Ohkubo, Y., Yokoyama, T. and Kanayama, C.: Clean Reversible Simulation of Ranking Binary Trees, Proc. *The 19th JSSST Workshop on Programming and Programming Languages*, (2016).

[3] Knott, G. D.: A Numbering System for Binary Trees, *Commun. ACM*, Vol.20, No.2, pp.113–115 (1977).

[4] Ruskey, F. and Hu, T. C.: Generating Binary Trees Lexicographically, *SIAM J. COMPUT*, Vol.6, No.4, pp.745–758 (1977).

[5] Er, M. C.: Enumerating Ordered Trees Lexicographically, *The Comput. J.*, Vol.28, No.5, pp.538–542 (1985).

[6] Bennett, C. H.: Logical Reversibility of Computation, *IBM J. Res. Dev.*, Vol.17, No.6, pp.525-532 (1973).

[7] Feng, X. and Park, S. (Eds.): Programming Languages and Systems, Axelsen, H. B. and Yokoyama, T.: *Programming Techniques for Reversible Comparison Sorts*, pp.407–426, Springer-Verlag (2015).

[8] Shamir, A.: Factoring numbers in $O(\log n)$ arithmetic steps, *Information Processing Letters*, Vol.8, pp.28–31 (1979).