可逆線形探索

2014SE024 家崎雄太　2014SE067 水野竣太郎
指導教員：横山哲郎

1　はじめに

可逆アルゴリズムは，出力を基にして入力を特定できるアルゴリズムである．なお，この性質は可逆性とよばれている．本来可逆アルゴリズムではないアルゴリズムでも，入力を特定するための情報を出力に追加する事によって，可逆性を与える事ができる．可逆性を与える動作は，可逆化と呼ばれている．

現在，可逆アルゴリズムに関する研究が行われている．例えば，文献 [2] ではソートアルゴリズムの可逆化が行われている．しかし，我々が知る限り，線形探索に特化した可逆アルゴリズムや可逆化の手段に関する議論が十分になされていないのが現状である．

線形探索に特化した効率的な可逆アルゴリズムを模索し，提案する事を本研究の目的として設定する．設定した目的を達成させるため，我々は Janus を使用して可逆線形探索を実装した．

Janus とは，命令型可逆プログラミング言語である．命令型非可逆プログラミング言語である C 言語と類似した構文を持つ．構文の類似性を利用して，本研究では線形探索の C 言語プログラムを Janus に書き換える事にした．

まず C 言語で線形探索を記述する．その C プログラムを埋め込み法と Bennett 法を用いて Janus に書き換える．その Janus プログラムを我々が提案した Janus プログラムと比較して，我々が提案した Janus プログラムがより効率的である事を示す．

2　線形探索と可逆化

本章では，非可逆な線形探索の概要を記述する，そして一般的に用いられている可逆化の手段に関する説明を行う．

```
1 int s_banpei(int data[],int next[],int prev[],
      int *tops,int key)
2 {
3     int f = 0;
4     while(data[*tops] != key){
5         int x = data[*tops];
6         spop(data,next,prev,tops,&x);
7     }
8     if(*tops != 0){
9         f = 1;
10    }
11    return f;
12 }
```

図 2.1　番兵を用いた線形探索：スタック

2.1　通常の線形探索

通常の線形探索は，探索の対象とするレコード列に変更を加える事なく探索できるという点で最も単純な線形探索アルゴリズムであるといえよう．通常の線形探索は，探したいキーを先頭のレコードが持つキーと比較して，キーの値が一致すればその時点で成功したものとして終了する．しかし，キーの値が一致しなければ，探索していたレコードが最後尾か否かを確認する．探索していたレコードが最後尾であれば，探索が失敗したものとして終了する．最後尾でなければ，次のレコードを探索の対象にする．通常の線形探索の長所は，レコード列に手を加えずに実行出来る事である．一方，短所は探したいキーと一致するキーを持つレコードが存在していなくても最後尾まで探索を続行しなければならない事である．

2.2　番兵を用いた線形探索

番兵を用いた線形探索は，レコード列の最後尾に探したいキーと一致する値のキーを持つ番兵と呼ばれるレコードを付け加えて探索を行う探索アルゴリズムである．番兵を用いた線形探索の長所は，キーの比較をする時点では最後尾かどうかを確認する必要がない事である．番兵を用いた線形探索の短所は，レコードを 1 つ追加するためのリソースが確保できなければ実装できない事である．

2.3　整列リストを用いた線形探索

整列リストを用いた線形探索は，探索の対象となっているレコード列を，キーが昇順あるいは降順に並ぶようにソートしたうえで探索するアルゴリズムである．ただし，本稿ではキーが昇順に並ぶようにレコード列をソートするものと仮定する．整列リストを用いた線形探索では，レコード列の最後尾に，∞ をキーに持つレコードを付け加えて先頭から探索する．探索しているレコードのキーが探したいキーより小さい値を持っているならば，次のレコードを探索の対象として続行する．探索しているレコードのキーが探したいキー以上の値を持っているならば，値が一致しているかどうかの確認を行う．キーの値が一致していれば，このアルゴリズムは成功して終了する．しかし，キーの値が一致していなければ，このアルゴリズムは失敗して終了する．整列リストを用いた線形探索は，探したいキーより大きい値のキーを持つレコードを探索せずに終了できるという長所を持つ．一方，整列リストを用いた線形探索の短所は，レコードをソートしなければならない事である．

2.4 ランダウアー法 (埋め込み法)

ランダウアー法は埋め込みと呼ばれる方法である．可逆性を保証するため，出力にゴミ情報を追加し，非可逆な計算により失われる情報を保存する．例えば C 言語での x=3 という代入は直前の x の値が特定できなくなるから非可逆である．Janus では x ^ =3 をする前に直前の値をゴミスタック g に push する．ゴミスタックは計算過程で生じたゴミ情報を保存するためのスタックである．push により代入の直前の値を保存することで可逆になる．埋め込みによる可逆シミュレーションの方法では時間的，空間的な面でトレードオフを持つ方法が広く研究されている．

2.5 Bennett 法

Janus における Bennett 法は call，結果のコピー，uncall によって行われる．call は関数を呼び出しすることが出来る．uncall は関数を逆呼び出しすることが出来る．Bennett 法は出力に入力したものを追加する．call により順方向の計算が呼び出されると，可逆性を保証するためにゴミ情報が出てくる．このとき uncall により関数を逆呼び出しすることで call により生じたゴミ情報を浄化することが出来る．

3 Janus による実装

Janus は命令型可逆プログラミング言語である．Janus は可逆性を保証する構文規則を持つ．Janus を利用することで可逆なプログラミングを記述することが出来る．Janus の変数宣言は int 型の変数，int 型の一次元配列と int 型のスタックを定義する．代入する際，+=（加算），-=（減算），^=（排他的論理和）を使用する．ただし代入式の左側に現れた変数は右側に現れてはいけない，すなわち x+=x という計算は行うことが出来ない．代入は変数の値を変える唯一の方法である．条件式 `if` e_1 `then` s_1 `else` s_2 `fi` e_2 について述べる．式 e_1 が真であるとき文 s_1 を実行する．文 s_1 を実行後，式 e_2 は真でなければならない．式 e_1 が偽であるとき文 s_2 を実行する．文 s_2 実行後，式 e_2 は偽でなければならない．繰り返し式 `from` e_1 `do` s_1 `loop` s_2 `until` e_2 について述べる．式 e_1 は最初，真でなければならない．後の繰り返しでは式 e_1 は偽でなければならない．最初に文 s_1 が実行される．式 e_2 が偽であれば文 s_2 を実行し，式 e_2 が真であれば実行を終了する．データ型スタックの操作について述べる．push(x,s) はスタック s に要素 x を追加し，x を 0 クリアする．pop(x,s) はスタック s から一番上の値を取り出し，x に格納する．local はローカル変数に記憶領域を割り当て，変数を式と同じ値で初期化する．local と delocal に囲まれた部分でしかその変数を扱うことができない．関数を呼び出すには call と uncall を用いる．call は関数を呼び出すことができる．uncall は関数を逆呼び出しすることが出来る．

3.1 C から Janus への書き換え

$$\mathcal{T}[\![\ \texttt{x = c;}\ ]\!] = \begin{array}{l}\texttt{push(x,g)}\\ \texttt{x \^{}= c}\end{array}$$

(a) 代入

$$\mathcal{T}[\![\ \texttt{int x = c;}\ ]\!] = \begin{array}{l}\texttt{local int x = c}\\ \quad\texttt{push(x, g)}\\ \texttt{delocal int x = 0}\end{array}$$

(b) 変数宣言

$$\mathcal{T}\left[\!\!\left[\begin{array}{l}\texttt{if(}\ e_1\texttt{)\{}\\ \quad s_1\\ \texttt{\}}\\ \texttt{else\{}\\ \quad s_2\\ \texttt{\}}\end{array}\right]\!\!\right] = \begin{array}{l}\texttt{if}\ \mathcal{T}[\![e_1]\!]\ \texttt{then}\\ \quad\mathcal{T}[\![s_1]\!]\\ \quad\texttt{push(1,g)}\\ \texttt{else}\\ \quad\mathcal{T}[\![s_2]\!]\\ \quad\texttt{push(0,g)}\\ \texttt{fi top(g) = 1}\end{array}$$

(c) 条件式

$$\mathcal{T}\left[\!\!\left[\begin{array}{l}\texttt{while(}\ e_2\texttt{)\{}\\ \quad s_1\\ \texttt{\}}\end{array}\right]\!\!\right] = \begin{array}{l}\texttt{push(1,g)}\\ \texttt{from top(g) = 1 loop}\\ \quad\mathcal{T}[\![s_1]\!]\\ \quad\texttt{push(0,g)}\\ \texttt{until}\ !\mathcal{T}[\![e_2]\!]\end{array}$$

(d) 繰り返し文 1

$$\mathcal{T}\left[\!\!\left[\begin{array}{l}\texttt{for (i = 0;}\ e_2\ \texttt{i++)\{}\\ \quad s_1\\ \texttt{\}}\end{array}\right]\!\!\right] = \begin{array}{l}\texttt{push(1,g)}\\ \texttt{from top(g) = 1 loop}\\ \quad\mathcal{T}[\![s_1]\!]\\ \quad\texttt{i+=1}\\ \quad\texttt{push(0,g)}\\ \texttt{until}\ !\mathcal{T}[\![e_2]\!]\end{array}$$

(e) 繰り返し文 2

図 3.3 C から Janus への書き換え規則

代入について C 言語では代入すると代入する前の値が何であったか分からなくなってしまう．Janus では x^=c の前にゴミスタック g に x の値を保存している．変数宣言について Janus では delocal int x = 0 で x が 0 でなければならないので，push(x, g) をすることで x を 0 クリアしている．if 文について Janus では可逆性を保証するためにアサーションを置く必要がある．アサーションとして top(g)=1 を置く．アサーションは if 節を実行した場合は真，else 節を実行した場合は偽でなければならない．アサーションを満たすために if 節の終わりで push(1, g)，else 節の終わりで push(0, g) を置く．繰り返し文について，C 言語の while 文は Janus の loop 文に書き換えられる．loop 文もアサーションが必要であるので top(g)=1 を

$prog \in$ Progs　$s \in$ Stms　$d \in$ Vdecs　$\odot \in$ ModOps
$p \in$ Procs　$e \in$ Exps　$t \in$ Type　$\otimes \in$ Ops
$q \in$ PIds　$x \in$ Vars　$c \in$ Cons

図 3.1 Janus の構文領域

$prog$ ::=	$P_{main}P^*$	*Janus* プログラム
d ::=	$x \mid x[c]$	スカラーと配列
t ::=	`int` | `stack`	データ型
P_{main} ::=	`procedure main() (int` d| `stack` $x)^*$ s)	$main$ 関数
p ::=	`procedure` $q(tx, ..., tx)$ s	関数定義
s ::=	x $\odot$`=` $e \mid x[e]$ $\odot$`=` $e \mid$	代入
	`if` e `then` s `else` s `fi` $e \mid$	条件式
	`from` e `do` s `loop` s `until` $e \mid$	繰り返し
	`push(`x`,`x`)` | `pop(`x`,`x`)` |	スタック操作
	`local` tx `=` e s `delocal` tx`=` $e \mid$	ローカル変数
	`call` q`(`x`,...,`x`)` |	関数呼び出し
e ::=	$c \mid x \mid x[e] \mid e \otimes e$ | `empty(`x`)` | `top(`x`)` | `nil`	式
c ::=	`-2147483648` | $\cdots$ | `0` | `1` | $\cdots$ | `2147483647`	定数
$\odot$ ::=	`+` | `-` | `^`	演算子
$\otimes$ ::=	$\odot$ | `*` | `/` | % | `&` | `|` | `&&` | `||` | `<` | `>` | `=` | `!=` | `<=` | `>=`	演算子

図 3.2 Janus 構文規則

置く．loop 文でのアサーションは繰り返し実行前は真，繰り返し実行中は偽でなければならない．それを満たすために `from` の前で `push(1, g)`，loop 節の終わりで `push(0, g)` をする．while 文の制御式 e_2 を Janus の `until` に e_2 の否定を制御式として置く．C 言語の for 文も loop 文で書き換えられる．while 文と同様，アサーションには `top(g)=1` を置く．for 文の制御式 e_2 も Janus の `until` に e_2 の否定を制御式として置く．

3.2 スタック

```
1  procedure spush(int data[],int next[],int prev
       [],int tops,int x)
2    local int i = 0
3      from i = 0 loop
4        i += 1
5      until data[i] = 0
6      data[i] ^= x
7      next[i] ^= -1
8      prev[i] ^= tops
9      next[tops] ^= -1
10     next[tops] ^= i
11     tops ^= prev[i]
12     tops ^= i
13   delocal int i = tops
```

図 3.4 push

配列である `data,next,prev` と整数型の `tops` を用いて，スタックを双方向リストで表現する．`data` はスタックに格納する値を表す．`next,prev` はスタックに格納されているキーの次，前の場所をそれぞれ表す．`tops` はスタックの一番上のキーの場所を表す．

図 3.4 のプログラムはスタックの一番下のキーを表す `data[0]` には元々値が格納されており，それ以外の配列には 0 が格納されているとする．その値の前後には値が格納されていないことを表すため `next[0]` と `prev[0]` には-1 を格納している．仮引数の `x` は格納するキーの値を表す．図 3.4 の 3–5 行の繰り返し文でキーが格納されていない配列を探し，6 行目で `data[i]` に格納したいキーを代入する．7 行目で格納したキーが一番上のキーであることを表すために `next[i]` に-1 を代入する．8 行目で `prev[i]` にそれまで一番上のキーであった場所を表す `tops` を代入する．11 行目で `tops` を 0 クリアして，12 行目で `tops` に今回 `spush` した値が一番上のキーであることを表すため，`tops` に `i` を代入する．

3.3 一般解法

今回はスタックを用いた線形探索について記述する．
図 3.5 のプログラムは一般解法である Landauer 法と Bennett 法を用いて作成した．仮引数には `data, next, prev,tops, key, f, g` を置く．ただしスタックの一番下には探したいキーと同じ値を番兵として格納する．入力は探索するキーの値が格納されている `data`，キーの前後の場所を格納した `prev, next`，スタックの一番上のキーの場所を表す `tops`，探したいキーを表す `key` である．出力は `f`(成功した場合 1，失敗した場合 0) と入力である．
図 3.5 のプログラムは 2 章の図 2.1 を図 3.3 の書き換え規則を用いて Janus で書き換えた．図 2.1 の 4–7 行の while 文は制御式として `data[*tops]!=key` を置く．この制御式の否定 `data[tops]=key` を図 3.5 の 3–9 行の繰り返し文の制御式として置く．

```
1 procedure s_banpei(int data[],int next[],int
      prev[],int tops,int key,int f,stack g)
2   push(1, g)
3   from top(g) = 1 loop
4     local int x = data[tops]
5       call spop(data,next,prev,tops,x)
6       push(x, g)
7     delocal int x=0
8     push(0, g)
9   until data[tops] = key
10  if tops != 0 then
11    push(f, g)
12    f ^= 1
13    push(1, g)
14  else
15    push(0, g)
16  fi top(g) = 1
17
18 procedure s_banpei_b(int data[],int next[],int
      prev[],int tops,int key,int f)
19  local stack g = nil
20    local int ff = 0
21      call s_banpei(data,next,prev,tops,key,ff,
            g)
22      f ^= ff
23      uncall s_banpei(data,next,prev,tops,key,
            ff,g)
24    delocal int ff = 0
25  delocal stack g = nil
```

図 3.5 番兵を用いた線形探索：一般解法

図 3.5 のプログラムの 3–6 行の繰り返し文で制御式として data[tops]=key を置く．これはスタックの一番上のキーが探したいキーと等しくなるまで繰り返すことを表している．図 2.1 の 8–10 行の if 文の制御式として*tops!=0 を置く．図 3.5 の 10–16 行の if 文の制御式として tops!=0 置く．tops=0 はスタックの一番上のキーが番兵であることを表している．スタックの一番上のキーが番兵でなければ，探索は成功したことを表し，f に 1 を代入する．
図 3.5 のプログラムは制御情報を中間のゴミ情報として保存する．スタックのキーが多くなるほどゴミ情報は大きくなり，中間ゴミは大きくなる．図 3.5 のプログラムは 2 パスであるので，実行時間は大きくなる．ステップ数は $14M+2S+8$ となる．ただし M は探索が成功した時，探したいキーが見つかった場所がスタックの上から何番目かを表す．失敗した時，探索するキーの個数を表す．S は探索が成功した時 1 を表し，失敗した時 0 を表す．

3.4 手動による解法

図 3.6 のプログラムは一般解法と異なり仮引数にスタック g を除いた data,next,prev,tops,key,f,data2 を置く．data2 には data,next が交互に格納されている．図 3.6 の 3–7 行の繰り返し文でアサーションとして tops=c を置く．5 行目で call spop(data,next,prev,tops,x) し，tops の値を変えていくことでアサーションを満たしている．図 3.6 の 6 行目の x は spop する前のスタックの一番上のキーを表す．x と等しい値が必要であるので data2 を用いた．

```
1 procedure s_banpei(int data[],int next[],int
     prev[],int tops,int key,int f,int data2[])
2   local int c = tops
3     from tops = c loop
4       local int x = data[tops]
5         call spop(data,next,prev,tops,x)
6       delocal int x = data2[2*data2[2*tops+1]]
7     until data[tops] = key
8     if tops != 0 then
9       f ^= 1
10    fi tops != 0
11    from data[tops] = key loop
12      local int x = data2[2*data2[2*tops+1]]
13        call spush(data,next,prev,tops,x)
14      delocal int x = data[tops]
15    until tops = c
16  delocal int c = tops
```

図 3.6 番兵を用いた線形探索：手動による解法

図 3.6 のプログラムは中間でゴミ情報を保存する必要がない．一般解法と異なり，代入する前にその値が何であったか保存する操作，if 文で if 節と else 節のどちらを実行したか保存する操作，繰り返し文のアサーションを満たすために push(1, g)，push(0, g) する操作が必要ない．また 1 パスであるので実行時間が小さくなる．ステップ数は $10M+S-1$ となる．

4 おわりに

本研究では線形探索 (通常の方法，番兵を用いた方法，整列リストと番兵を用いた方法) の配列版，スタック版について記述した．C 言語で書いた線形探索を一般的な方法である埋め込み法，Bennett 法を用いて Janus に書き換えた．一般的な方法でかかれたプログラムと自分たちが提案したプログラムの比較をした．一般解法では中間でゴミスタックにゴミ情報を保存する必要がある．今回注目した中間のゴミ，ステップ数の点では効率化出来た．

参考文献

[1] Knuth,K.E：“The Art of Computer Programming Volume 3 Sorting and Searching Second Edition 日本語版”, 株式会社 KADOKAWA（2015）

[2] Axelsen ,H.B., Yokoyama ,T.: Programming Techniques for Reversible Comparison Sorts. In: Feng X., Park S. (eds) Programming Languages and Systems. Lecture Notes in Computer Science, vol 9458. Springer, Cham(2015)

[3] Yokoyama,T.:Reversible Computation and Reversible Programming Languages,Electronic Notes in Theoretical Science253,pp.71-81(2010).