可逆探索アルゴリズム

M2018SE009 増田大輝
指導教員：横山哲郎

1 はじめに

現在，非可逆なアルゴリズムを，入出力が等しい可逆アルゴリズムに変換する可逆シミュレーションを行う（以下可逆化）一般解法として Bennett 法 [5] が知られている．可逆シミュレーションとしての可逆アルゴリズムは，一般解法によって作られた非可逆なアルゴリズムと入出力が等しい可逆アルゴリズムに比べて，実行時間やメモリ使用量，ゴミ出力量が小さいものが存在する場合がある．

例えば，比較ソートアルゴリズムの可逆化は，漸近的計算時間とごみ出力が最適である可逆シミュレーションが提案されている [4]．また，線形探索アルゴリズムについての可逆化は，何を入出力と見なすかや，探索の成功失敗によってプログラムの走査回数に変化が生じた [5]．これは，非可逆なアルゴリズムでは現れなかった変化である．しかし，我々の知る範囲において，データ構造や入出力の違いで最適な可逆アルゴリズムにどのようなトレードオフ関係があるのかを述べた文献は存在しない．

本稿では，線形探索，木構造の探索，グラフ構造の探索アルゴリズムの可逆化を行う．また，探索アルゴリズムを可逆化した際にメモリ使用量や計算量に関するどのようなトレードオフを解析する．これは，他の探索アルゴリズムを解析する際に，非可逆なアルゴリズムでは現れなかったトレードオフを考慮する必要性の有無を判明する一つの方法を示している．

文献 [3] では，Bennett 法よりも時間，空間に最適な可逆線形探索アルゴリズムを示している．しかし，なにを入出力とするかや，データ構造の違いが最適な可逆線形探索アルゴリズムの入力を走査する回数やメモリ使用量などに影響を与えることは書かれていない．

可逆線形探索アルゴリズムでは，出力データの種類やデータ構造の種類により，時間的・空間的計算量に変化が表れることを示す．また，それらを考慮したうえでアルゴリズムにどのようなトレードオフが生じるのかを解析する．

木構造の可逆探索アルゴリズムでは，深さ優先探索と幅優先探索アルゴリズムの効率的な可逆アルゴリズムを設計し，一般解法よりも様々な指標で優れているアルゴリズムを開発する．

グラフ探索アルゴリズムの可逆化では，木構造の可逆探索アルゴリズムを比較することにより，閉路によって時間・空間計算量がどの程度変化するのかを解析する．

研究によって判明したトレードオフ関係はアルゴリズムを可逆化する際に注意すべき新たな指標であり，今後，他のアルゴリズムの可逆化に応用されることが期待される．また，探索アルゴリズムは，ソートアルゴリズムと同様に多くのアルゴリズムで利用されている．最適な可逆探索アルゴリズムを設計・実装することは，より複雑な可逆アルゴリズムを設計する際に役立つことが予想される．

本稿のプログラムは，`http://tetsuo.jp/janus-playground/` にあるオンラインインタプリタから実行可能である．

2 関連研究

2.1 可逆性

可逆計算とは，計算システムにおいて，そのシステムの初期状態と最終状態を除いたすべての状態が直前と直後の時刻の状態を一意に定めることができる性質，つまりシステムの動作を時間軸の順方向のみでなく，逆方向においても辿ることのできる性質である [2]．

2.2 可逆ソートアルゴリズム

ソートアルゴリズムとは，レコード列をそのキーの全順序関係によって並べるアルゴリズムである．可逆ソートアルゴリズムは一般的な可逆化 [5] によって生成されるものよりも，漸近的空間・時間に計算量が効率的なものが知られている [4]．

2.3 双方向リスト

双方向リストとは，リスト構造の一種で，一般的にリストと呼ばれている単方向リストが次データへのポインタのみをもつのに対して，次データと前データへの両方のポインタをもつリスト構造である．双方向リストの利点は，要素の追加・削除に必要な時間が $\mathrm{O}(1)$ であるという点である．これは，前データへのポインタによって追加・削除する要素の直前の要素を $\mathrm{O}(1)$ の時間で辿ることができるからである．

可逆アルゴリズムでは，計算を逆実行する際に直前の要素を辿る場合がある．その処理にかかる時間を双方向リストであれば $\mathrm{O}(1)$ で可能である．したがって，可逆計算でリストを用いる場合は双方向リストを用いることが多い．

双方向リストを配列で表現する．各要素を格納する `head`，次のレコードへのポインタである `next`，前のレコードのポインタである `prev` の 3 つの配列を用意する．例えば $\mathtt{head}[3] = \{2,4,6\}$，$\mathtt{next}[3] = \{2,-1,1\}$，$prev[3] = \{-1,2,0\}$ とし，-1 は $NULL$ とする．$\mathtt{prev}[1] = 2$ より，$\mathtt{head}[1]$ の直前の要素は $\mathtt{head}[2]$ である．また，$\mathtt{next}[0] = 2$ より，$\mathtt{head}[0]$ の次の要素は $\mathtt{head}[2]$ である．$\mathtt{prev}[0] = -1$ より，$\mathtt{head}[0]$ はリストの先頭の要素である．$\mathtt{next}[1] = -1$ なので，$\mathtt{head}[1]$ はリストの末尾の要素である．

2.4 一般解法によるアルゴリズムの可逆化

現在，アルゴリズムを可逆化する一般解法は複数存在する．本稿では，知名度の高い一般化手法の一つである Bennett 法 [5] を説明する．

```
1 procedure sercl(int head[], int next
    [], int k, int l, stack g)
2 from l = 0 loop
3     local int t = next[l]
4     push(l,g)
5      l ^= t
6     delocal int t = l
7 until head[l] = k || next[l] = -1
8 if head[l] != k then
9     push(l,g)
10    l ^= -1
11 fi l = -1
```

図 1 リストを用いた可逆線形探索

Bennett 法とは埋め込み法と呼ばれる計算過程で発生する情報を出力として保存する可逆化手法を用いた一般解法である．埋め込み法では失われるはずの情報を一時保存する．最終的に保存した情報はごみとして消去するのでエネルギーを消費するという欠点がある．

Bennett 法では，埋め込み法での計算で得られた出力を別の変数に格納し，埋め込みの逆計算を行うことによって保存してある入出力以外の情報を消去する．よって，計算過程で保存した入出力以外の情報をなくすことができる．しかし，Bennett 法では計算と逆計算を行うので，プログラムを 2 回走査する必要がある．したがって，計算時間も増えてしまう．また，最終的に不必要な情報を消去することができるが，計算過程で情報を保存するので，メモリ使用量は埋め込みと同様に大きくなってしまう．

3 可逆線形探索アルゴリズム

線形探索とは，ファイルと呼ばれる n 個のキーをもつ要素のレコード列を，先頭から末尾まで順に比較を行い，与えられた気を含むレコードを見つけることである．線形探索の漸近的計算量は $\mathrm{O}(n)$ であり，レコードの数に比例する．

本稿では，データ構造は，配列，リスト（データの書き換えの有無），双方向リストの 4 つを考える．線形探索の出力として，キーをもつレコードの有無を示すフラグ，キーをもつレコードの位置，キーをもつレコードの個数を考える．今回は，可逆化によって探索アルゴリズムに新たな影響を与えた組み合わせのみを記載する．可逆アルゴリズムの解析対象として

- 元の出力以外のメモリ使用量　M_1
- 元の入出力以外のごみ出力量　M_2

を考える [5]．

本稿では，解以外に可逆アルゴリズムで返される出力をごみ出力とする．本章では，M_2=0 の場合を考える．

3.1 データ構造がリストの場合

リストを用いた場合，停止した位置の情報が一意に定まらないので，探索の成否に関わらず走査回数が 2 必要である．

```
1 procedure srchd(int head [], int next
    [], int prev [], int k, int l)
2  from l = 0 loop
3  local int t = next [l]
4  l ^= prev [t] ^ t
5  delocal int t = l
6 until next [l] = -1 || k = head [l]
7
8 procedure srchdopt (int head [], int
    next [], int prev [], int k, int n,
     int f)
9  local int l = 0
10 call srchd (head ,next ,prev ,k,l)
11 if l = n then
12 l ^= n // zero clear
13 else
14 f ^= 1
15 uncall srchd (head ,next ,prev ,k, l)
16 fi f != 1
17 delocal int l = 0
```

図 2 双方向リストを用いた可逆線形探索

ROM のようにデータの書き換えができない場合，リストを辿るのにレコード列 n 以下の位置情報を最終的に消去する必要があるので，M_1 は $\mathrm{O}(n)$ 必要である．RAM のようにリストがミュータブルの場合は，探索時にリストのポインタをつなぎ替えることによって，辿ってきたレコードのポインタを上書きして保持することで，M_1=$\mathrm{O}(1)$ で実現可能である．

例えば，図 1 の sercl は，リストを用いてファイルを表しレコード位置を返す可逆線形探索である．k が入力の 1 つであるキーで，l が出力となるレコードの位置を格納する．リストは，レコード head[]，次のレコードへのポインタ next[] の 2 つの配列を用いたリストの添え字表現で表す．簡単のため，リストの先頭要素は添字 0 に格納されているとする．また，− 1 はどの要素も指さない *NULL* ポインタを表し，探索が失敗した場合は *NULL* ポインタである-1 を返す．sercl の本体では，リストの先頭から末尾まで順に探索を行い，キーをもつレコードを発見する，または，末尾まで探索を行った場合に停止する．

3.2 データ構造が双方向リストの場合

双方向リストの場合，リストを辿るのに必要な位置情報は前のレコードを指すポインタによって消去することができる．よって，M_1=Θ (1) で実現可能である．走査回数は，探索が成功して停止したのかを保持している情報から判断することができないので 2 必要である．

末尾のレコードのポインタの値をあらかじめ保持している場合，走査回数 1 で実現可能となる場合もある．

例えば，図 2 の srchd は，双方向リストを用いてフラグを返す可逆線形探索であり，末尾のレコードを指すポ

```
1 procedure srchf1 (int k[], int key ,
      int n, int f, int i)
2  from i = 0 loop
3  i += 1
4 until i = n-1 || k[i] = key
5  if k[i] = key then
6  f ^= 1
7 fi k[i] = key
8
9 procedure srchf1opt(int k[], int key ,
       int n, int f)
10     local int i = 0
11     local int ff = 0
12     call srchf1(k,key,n,ff,i)
13     f ^= ff
14     uncall srchf1(k,key,n,ff,i)
15     delocal int ff = 0
16     delocal int i = 0
```

図 3 配列を用いたフラグを返す可逆線形探索

```
1 procedure srchp1 (int k[], int key ,
      int n, int f, int i)
2  from i = 0 loop
3  i += 1
4 until i = n-1 || k[i] >= key
5  if k[i] = key then
6  f ^= -1
7  f ^= i
8 fi k[i] = key
9
10 procedure srchp1opt(int k[], int key ,
       int n, int f)
11     local int i = 0
12     call srchp1(k,key,n,f,i)
13     if f = -1 then
14         uncall srchp1(k,key,n,f,i)
15     else
16         i ^= f
17     fi f = -1
18         delocal int i = 0
```

図 4 整列された配列を用いたレコード位置を返す可逆線形探索

インタの値を保持しているとする．仮引数は，レコードである head，次のレコードを指すポインタである next[]，前のレコードを指すポインタである prev[]，キー情報である k，探索するポインタを格納する l である．先頭のレコードを指すポインタである l=0 を読み込んでループを開始する．次のレコードを指すポインタを格納する引数 t を生成し，4 行目で l を次のレコードを指すポインタにアップデートする．その後，引数 t の値を l を用いて消去する．この工程を探索が成功するか，末尾のレコードを探索するまで繰り返す．srchdopt では，l の代わりにフラグを格納する f と，末尾のレコードの値を格納する n を仮引数とする．srchdopt では，srchd を呼び出し，探索の成否によって場合分けを行う．探索が成功した場合，フラグ f に 1 を格納し，l を srchd を逆呼び出しすることで消去する．探索が失敗した場合，末尾のレコードで探索が終了しているので，n を用いて l を消去する．よって，成功時には操作が 2 必要であるが，失敗した場合は 1 で実現可能である．

3.3 データ構造が配列の場合

出力がフラグの場合は，探索が成功した場合には計算を巻き戻して位置情報を消去するので，走査回数が 2 回必要だが，探索が失敗した場合，計算を巻き戻さず，末尾の情報を利用して位置情報を消去することができるので，走査回数は 1 回で可能である．

例えば，srchf1opt であれば，探索成功時はプロシージャsrchf1 を逆実行するが，探索失敗時には末尾のレコード位置である $n-1$ を用いて i を消去している．

レコード列が整列されている場合，末尾のレコードに辿り着くことなく探索を終えることがあるので，探索失敗時も成功時と同様に走査回数が 2 回必要になる．

出力が条件を満たしたレコードの位置情報の場合，探索の成否によらず走査回数が 1 回で可能である．探索成功時は位置情報を出力によって消去することができ，失敗時は末尾の情報を利用して位置情報を消去することができるからである．

レコード列が整列されている場合，末尾のレコードに辿り着くことなく探索を終えることがあるので，探索失敗時に走査回数が 2 回必要になる．srchp1opt では，探索成功時は i に格納されている数字を f を用いて消去している．探索失敗時には逆実行を行うことによって i を消去する．

出力がキーをもつレコードの個数の場合，常に走査回数を 1 にすることができる．なぜなら，個数を出力とする場合にはすべてのレコードを探索する必要があるので，停止する位置が必ず末尾のレコードとなるからである．よって，実行時間は通常の $\mathrm{O}(n)$ ではなく $\Theta\ (n)$ となる．

4 木構造の探索アルゴリズムの可逆化

木とは，閉路をもたない単純で連結な有向グラフのことである．木構造の，要素部分を節点といい，各要素は親子関係をもつ．子のない末端の節点を葉，親のない先頭の節点を根といい，要素同士のつながりを枝という．本稿では，簡単のため二分木を扱う．木構造の探索法には，幅優先探索と深さ優先探索がある．幅優先探索は，根に近い節点から順に調べていく探索法である．深さ優先探索は，目的の節点が見つかるか葉に行きつくまで探索を続け，最も近くの探索の終わっていない節点まで戻り，同様に探索を行う方法である．

4.1 幅優先探索

ヒープ木のように，根を先頭とし，根から順に配列に格納したオブジェクトとみなすと線形探索と同様の計算手順で探索可能である．その場合の計算時間は $O(n)$ で，M_1=Θ (1),M_2=Θ (0) である．

4.2 深さ優先探索

アルゴリズムの各ステップで，探索中の位置情報のみから，そのステップの直前にどこを探索していたのかを一意に定めることができない．したがって，直前に探索した位置情報を保存する必要がある．

我々は，以下の特徴に着目して効率的な可逆化を試みる．根から探索を開始し，左の子があればそれを次に探索する．その際に，探索した方向を記憶する．方向の情報と位置情報の 2 つを用いることにより，直前の位置情報を一意に定めることができる．また，探索が終了して実行を終了するのは必ずすべてのレコードを探索後の根のレコードとなるようにする．こうすることによって，探索が成功して実行を終了した地点の位置情報を一意に定めることができる．

本章では，配列を用いた疑似リストと探索するキーを入力とし，探索が成功した場合に 1 を，失敗した場合に 0 を出力する深さ優先探索を考える．効率的な（非可逆の）深さ優先探索を可逆化する一般解法 [5] は，実行時間が $mO(n)$ となるが，ごみ出力も $O(n)$ 必要になるので，M_2=0 にする場合は入力ファイルを 2 回走査する必要がある．

提案する木構造の可逆深さ優先探索プログラムを図 4 に示す．簡単のため，レコードはキーのみを含むものとし，キーの列は配列 $in[][0]$ に格納されているものとする．また，木構造は二分木とする．配列 $in[][1]$ は左の子を示すポインタ，配列 $in[][2]$ は右の子を示すポインタ，探索するキーを k，出力を格納する変数を f とする．f 以外の配列と変数の値は呼び出しの前後で変化しない．

可逆プロシージャ`func` では，主に可逆再帰呼び出しを行う．まず，探索地点が $NULL$ を示す`-1` でないことを確認する．`4,5` 行目の再帰呼び出しにより次に探索する要素を決定する．左に子があれば `4` 行目を実行し，探索地点を左の子に移す．左の子がない，または，左の子が探索済みである場合，右に子があれば `5` 行目を実行し，探索地点を右の子に移す．左右両方の子がない場合，または，どちらも探索済みの場合は親の要素に戻る．キーをもつ要素を見つけた場合，f に `1` を格納する．この工程をすべての要素を探索し，根の要素に戻ってくるまで繰り返す．

`func1` は，不要な出力である `visit` を消去する可逆プロシージャである．`visit` は根から始まり，根で終わるので根の添え字である `0` で消去することができる．

提案プログラムでは，$M_2 = 0$ の制約があっても，探索の成否に関わらず走査回数を 1 に抑えることに成功している．元の出力以外のメモリ使用量 M_1 は，局所的に定義した変数のみの Θ (1) である．実行時間は常に根の要素で停止するので，Θ (n) である．よって，時間計算量では一般解法の `mO(`n`)` に劣る．

```
1  procedure func(int in[][], int k, int
       f, int visit)
2      if visit != -1 then
3          local int temp = visit
4          call func(in, k, f, in[temp
               ][1])
5          call func(in, k, f, in[temp
               ][2])
6          if in[temp][0] = k then
7              f ^= 1
8          fi in[temp][0] = k
9          delocal int temp = visit
10     fi visit != -1
11
12 procedure func1(int in[][],int k, int
       f)
13     local int v = 0
14     call func(in,k,f,v)
15     delocal int v= 0
```

図 5 再帰関数を用いた木構造の可逆深さ優先探索

5 おわりに

可逆線形探索では，入力データの構造や出力データの変化が入力データを走査する回数やごみ出力量に影響を及ぼすことを明らかにした．また，探索アルゴリズムのように成否の出るアルゴリズムの可逆化では，その成否によって計算量が変化する可能性があることが判明した．これらの影響は，非可逆なアルゴリズムでは起こりえなかった現象であり，可逆アルゴリズムにおいて考慮しなければならない新しい要因を示した．このことから，可逆アルゴリズムの最適化では，通常のアルゴリズムの最適化ではあまり影響を及ぼすことのなかった要因までも考慮に入れて研究を行う必要があるといえる．

この知見は可逆探索アルゴリズム特有の現象ではなく，より複雑な可逆アルゴリズムでも考慮しなければならない影響である．よって我々は，可逆アルゴリズムの解析を行う際に必要となる知見を得ることができたといえる．

木構造の深さ優先探索の可逆化では，すべての面で一般解法よりも優れた可逆アルゴリズムを設計することはできなかった．しかし，入力の走査回数やメモリ使用量の点で，独自の手法を用いることで一般解法より優れた面をもつ解法を提案した．可逆な木構造の深さ優先探索を利用する際に選択肢を増やすことに成功したといえる．

6 今後の課題

6.1 木構造の双方向リストを用いた可逆探索アルゴリズム

単方向リストを用いて効率的な深さ優先探索アルゴリズムを設計・実装した．可逆線形探索で用いた双方向リストを利用して効率的な深さ優先探索を設計・実装した

場合どのようなトレードオフが発生するか調べる．

6.2 グラフ構造の探索アルゴリズムの効率的な可逆化

論文 [6] において行われているグラフ構造の効率的な深さ優先探索アルゴリズムを設計，実装する．

6.3 木構造とグラフ構造の可逆探索アルゴリズムの比較

木構造とグラフ構造の相違点は閉路の有無である．閉路の有無によって可逆探索アルゴリズムの最適な設計にどのような影響を与えるかを解析する．

参考文献

[1] Michael, P.F.:The Future of Computing Depends on Making It Reversible, available from(https://spectrum.ieee.org/computing/hardware/the-future-of-computing-depends-on-making-it-reversible) (accessed 2017-08-25).

[2] 萩谷昌己，横森貴（編者）：可逆計算，森田憲一：ナチュラルコンピューティング・シリーズ，近代科学社（2012）.

[3] 家崎雄太，水野竣太郎：可逆線形探索，南山大学 2017 年度卒業論文（2018）.

[4] Tetsuo, Y. and Holger, B.A.:Programming Techniques for Reversible Comparison Sorts, *Programming Languages and Systems, pp.407-426(2015).*

[5] Bennett, C.H.:Logical reversibility of computation, IBM Journal of Research and Development, vol.17, no.6, pp.525-532(1973).

[6] 浅野早紀，山口春樹：可逆な深さ優先探索，南山大学 *2018* 年度卒業論文，*pp.21-26(2019).*