効率的な可逆線形探索と木構造の可逆深さ優先探索の設計と解析

M2018SE009 増田大輝
指導教員：横山哲郎

1 はじめに

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一般解法である Bennett 法 [2] が知られている．しかし，汎用性のある一般解法を用いず，アルゴリズム固有の問題を解決することで可逆プログラムを作成する方法もある．このような可逆プログラムは，入出力が等しい一般解法によって作られた可逆プログラムに比べて，実行時間やメモリ使用量，ごみ出力量が小さいものが存在する場合がある [1]．Frank は効率的な可逆探索アルゴリズムの空間計算量を示したが，効率的な可逆アルゴリズムの設計・実装についての議論は未だ十分に行われていない [3]．また，個々の効率的な可逆アルゴリズムの設計・実装は並列計算をするうえで有用であり，並列離散事象シミュレーションなどに応用されている．また，探索アルゴリズムは整列アルゴリズムと並び，様々なアルゴリズムに組み込まれているアルゴリズムであり，探索アルゴリズムの研究による知見は多くのアルゴリズムに応用することができるであろうと期待される．

本稿では，効率的な可逆アルゴリズムを設計するための新たな知見を得るという目的のため，可逆線形探索，木構造の可逆深さ優先探索の設計・実装を行う．また，実装した可逆探索プログラムのメモリ使用量や計算量に関する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があることを示す．

本研究では上で挙げた目的を達成するために以下の研究課題を設けた．

RQ1．入力ファイルのデータ構造や出力が可逆線形探索に影響を及ぼすのか解析する．

RQ2．ごみ出力が 0 であるという条件下で効率的な可逆線形探索の時間・空間計算量にトレードオフ関係が発生するのか解析する．

RQ3．Bennett 法を用いて作成した木構造の可逆深さ優先探索よりも時間・空間計算量のどこかで優れているものを設計・実装する．

2 関連研究

2.1 可逆プログラミング言語 Janus

Janus の構文について説明する．可逆代入 x `^=` e は，e に x を含んではならないという構文上の制約がある．この制約下での x `^=` e は，変数 x と式 e の値の排他的論理和を x に格納する．`push`(x,g) はスタック g に変数 x の値を格納して x を空にする．`top`(g) はスタック g の頂点の値を返す．`local int` le `=` $init_1$ は，$init_1$ の値を初期化するために，左辺式 le で表される変数や配列の領域を確保する．`delocal int` le `=` $init_2$ は，le で表される変数や配列が $init_2$ の値であることを確認し，その領域を解放する．le_1 `<=>` le_2 は，両辺の値を交換する．条件 `if` e_1 `then` s_1 `else` s_2 は，テスト e_1 が真であるとき s_1 を実行し，アサーション e_2 が真であることを確認する．e_1 が偽であるならば s_2 を実行し，アサーション e_2 が偽であることを確認する．繰り返し `from` e_1 `do` s_1 `loop` s_2 `until` e_2 は，アサーション e_1 が真であることを確認し，テスト e_2 が真になるまで s_2 を実行する．その後，e_2 が偽になることを確認して s_1 を実行することを繰り返す．この繰り返し中で e_1 が真になった場合は異常終了する．プロシージャ呼び出し `call` p(${e_1}$,...,${e_n}$) は，プロシージャpの本体の文を順に実行する．プロシージャの逆呼び出し `uncall` p(${e_1}$,...,${e_n}$) は，プロシージャp の本体の文を逆変換したものを実行 (逆実行) する．

2.2 可逆アルゴリズムの一般解法

現在，可逆アルゴリズムを設計する一般解法は複数存在する．本稿では，知名度の高い一般解法の一つである埋込み法 [2] と Bennett 法 [4] を説明する．

埋込み法とは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．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 法とは，埋込み法等の方法で作られた可逆プログラムの実行後に，計算で得られた出力を別の変数に格納し，埋込みの逆実行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一般解法である．本稿では，可逆プロシージャh に対して Bennett 法を適用して得られた可逆プロシージャを $B(h)$ と表すことにする．

3 可逆線形探索

線形探索とは，ファイルと呼ばれる n 個のキーをもつレコード列を，先頭から末尾まで順に比較し，キーを含むレコードを見つけるアルゴリズムである．線形探索の漸近的計算量は $\mathrm{O}(n)$ であり，レコード数に比例する．

本稿では，データ構造は，配列，リスト，双方向リストの 3 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，キーの値と等しいレコードの位置（以下レコード位置）を考える．可逆アルゴリズムの解析する上での指標として

- 元の入出力以外のメモリ使用量　M
- 元の入力以外のごみ出力量　G

を考える．

本稿では，解と入力以外の出力をごみ出力とする．

図 2 は図 1 を埋込み法を利用して作成した線形探索プログラムである．可逆プログラムでは，C 言語のときの入力である `r`，`n`，`k` に加え，出力の値を格納する `f` と計算途中の情報を保存するスタック `g` を入力とする．

```
1 int srchf(int r[], int n, int k)
2 {
3   int f = 0;
4   int i;
5   for (i = 0; i<n && f==0; i++)
6     if (r[i] == k)
7       f = 1;
8   return f;
9 }
```

図 1 出力がフラグの線形探索

```
1 procedure srchf(int r[], int n, int k,
      int f, stack g)
2   f ^= 0
3   local int i = 0
4     push(1,g)
5     from top(g) = 1 loop
6       if r[i] = k then
7         push(f,g)
8         f ^= 1
9         push(0,g)
10      else
11        push(0,g)
12      fi top(g) = 1
13      i += 1
14      push(0,g)
15    until !((i < n) && (f = 0))
16    push(i,g)
17  delocal int i = 0
```

図 2 埋込み法を用いた可逆線形探索

埋込み法により，C プログラムの 4 行目は Janus プログラムの 3 と 16–17 行目，5 行目は 3–4 と 13–15 行目，6–7 行目は 6–12 行目に変換される．

非可逆な線形探索では漸近的なメモリ消費量とごみ出力量は $O(n)$ である．埋込み法を用いて作成した可逆プログラムは計算途中の情報をすべてスタックに保存するので元となった非可逆プログラムとメモリ消費量やごみ出力量が等しい．

図 3 は入力ファイルが配列，出力がフラグの可逆線形探索プログラムである．仮引数は C 言語の `r`，`n`，`k`，`f` に加えて探索位置の変更を行うポインタとして `g` を用いる．先頭のレコードを指すポインタである `g` = 0 を読み込んでループを開始する．2–4 行の条件文で探索しているレコードがキーと等しい場合，f に 1 を格納する．その後アサーションとして条件文が真であったとき，`r[g] = k` を満たしていることを確認する．5 行目でポインタを次のレコードの位置に更新する．この工程を 6 行目の繰り返し文の制御式 `g >= n || f != 0` が満たされるまで繰り返す．

```
1 procedure srchf_opt(int r[], int n,
    int k, int f, int g)
2   from g = 0 do
3     if r[g] = k then
4       f ^= 1
5     fi r[g] = k
6     g += 1
7   until g >= n || f != 0
```

図 3 手動で作成した可逆線形探索（配列，フラグ）

漸近的な計算時間は $O(n)$ であり，非可逆な線形探索の場合と等しいが，元の入出力以外のメモリ使用量と元の入力以外のごみ出力量は $O(1)$ である．

```
1 procedure srchl_opt(int r[], int n,
    int k, int f)
2   f ^= -1
3   local int i = 0
4     from i = 0 do
5       if r[i] = k then
6         f ^= (-1) ^ i
7       fi f = i
8     loop
9       i += 1
10    until i >= n-1 || f != -1
11  delocal int i = (f + n) % n
```

図 4 手動で作成した可逆線形探索（配列，レコード位置）

図 4 は入力ファイルが配列，出力がレコード位置の可逆線形探索プログラムである．漸近的な計算時間は $O(n)$ であり，非可逆な線形探索の場合と等しい．探索が終了した地点を出力から一意に定めることができるので，元の入出力以外のメモリ使用量は $O(1)$ だが，元の入力以外のごみ出力量は 0 で実行可能である．

```
1 procedure srchlist_opt(int f, int head
    [], int next[], int k, int g)
2   from f = 0 loop
3     f <=> g
4     f <=> next[g]
5   until f = -1 || k = head[f]
```

図 5 手動で作成した可逆線形探索（リスト，レコード位置）

図 5 は入力ファイルがリスト，出力がレコード位置の可逆線形探索プログラムである．先頭のレコードを指すポインタである $g = 0$ を読み込んでループを開始する．2 行目で引数 `f` と `g` の値を入れ替える．3 行目でポインタを次のレコードの位置に更新する．この工程を 4 行目の繰り返し文の制御式 `f = -1 || k = head[f]` が満たされ

るまで繰り返す．漸近的な計算時間は O(n) であり，非可逆な線形探索の場合と等しい．元の入出力以外のメモリ使用量は O(1) だが，リストのポインタを表す $next$ に格納されている値が実行の前後で変化する可能性があるため，ごみ出力量は O(n) 必要になる．

```
1 procedure srchdl_opt(int f, int head
    [], int next[], int prev[], int k,
    int g)
2   from f = 0 loop
3     f <=> g
4     f ^= next[g]
5     g ^= prev[next[f]]
6   until f = -1 || k = head[f]
```

図 6　手動で作成した可逆線形探索（双方向リスト，レコード位置）

図 6 は入力ファイルが双方向リスト，出力がレコード位置の可逆線形探索プログラムである．先頭のレコードを指すポインタである $f = 0$ を読み込んでループを開始する．2 行目で引数 f と引数 g の値を入れ替える．3 行目で f を次のレコードを指すポインタに更新する．その後，引数 g の値を f を用いて消去する．この工程を探索が成功するか，f の値が -1 つまり $NULL$ になるまで繰り返す．漸近的な計算時間は O(n) であり，非可逆な線形探索の場合と等しい．元の入出力以外のメモリ使用量とごみ出力量は直前のレコードの位置を示すポインタである prev を利用することで O(1) で実行可能である．

表 1　可逆線形探索のメモリ使用量の比較

プログラム	メモリ使用量 M	ごみ出力量 G
srchf	O(n)	O(n)
B(srchf)	O(n)	—
srchf_opt	O(1)	O(1)
srchl	O(n)	O(n)
B(srchl)	O(n)	—
srchl_opt	O(1)	—
srchlist	O(n)	O(n)
B(srchlist)	O(n)	—
srchlist_opt	O(1)	O(n)
srchdl	O(n)	O(n)
B(srchdl)	O(n)	—
srchdl_opt	O(1)	O(1)

表 1 は実装した可逆線形探索を比較した表である．—は入力が大きい場合は 0 に漸近していることを示す．可逆線形探索のメモリ使用量やごみ出力量は，何を入出力とするのかや，入力データのデータ構造によって漸近的にも異なった．Bennett 法を使用せず G が 0 で実行可能であった可逆線形探索プログラムは，入力データの構造が配列，出力がレコード位置の srchl_opt のみであった．また，手動で作成した可逆線形探索プログラムの中で，データ構造がリストである srchlist_opt のみが G が O(n) であった．これらの結果から，可逆線形探索は非可逆なアルゴリズムではあまり考慮されなかった要素である出力の種類や入力データのデータ構造の種類によっても計算複雑度が変化することが判明した．

探索アルゴリズムには探索の成功，失敗がある．非可逆な探索アルゴリズムでは探索の成否が実行時間に大きな影響を及ぼすことは少なかったが，可逆探索では，探索の成否がによって入力ファイルの走査回数が変化する場合がある．元の入出力以外のごみ出力量が 0 であるという条件下で効率的な可逆探索の設計する場合は探索の成否まで考慮に入れなければならないことを示す．

```
1  procedure fsrch(int r[], int n, int k,
     int f)
2    local int i = 0
3      local int ff = 0
4        call srchf_opt(r, n, k, ff, i)
5        f ^= ff
6        if f = 1 then
7          uncall srchf_opt(r, n, k, ff,
             i)
8        else
9          i ^= n-1
10       fi f = 1
11     delocal int ff = 0
12   delocal int i = 0
```

図 7　データ構造が配列で出力がフラグの可逆線形探索

図 7 の fsrch は，配列を用いてファイルを表しフラグを返す可逆線形探索プログラムである．仮引数は図 3 の srchf_opt と同名のものは同じ用途である．fsrch の本体では，srchf_opt が呼び出され，探索の成否によって場合分けが行われる．探索に成功した場合，探索を終了した場所の情報を srchf_opt を逆実行することによって消去する．探索が失敗した場合，探索の終了地点は常に配列の末尾であるのでレコードの個数 n の情報を用いて探索を終了した地点の情報を消去する．探索の成否が入力ファイルの走査回数，つまり実行時間に影響を及ぼすことは非可逆な探索では起こりえなかったことで，可逆探索特有のことである．

4　木構造の可逆深さ優先探索

深さ優先探索とは，根から順に左部分木から右部分木まで外周を辿りながら節点を順に探索するアルゴリズムである．本稿では，簡単のため二分木を扱い，各節点はキーの情報のみをもつこととする．深さ優先探索では，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がごみ出力となる．

我々は，以下の特徴に着目して効率的な可逆探索の設計

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 つを用いて，直前の位置情報を一意に定める．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，探索が終了した地点の位置情報を一意に定める．

今回，提案した設計方法での実装を行うことができなかった．提案した方法では探索した方向の更新を行う際の直前の方向の情報を消去することができなかった．よって，実装では探索方向だけでなく木全体を保存することによって可逆性を保持したまま探索を行うように実装した．

実装した木構造の可逆深さ優先探索プログラムを図 8 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 `st` 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．左の子がないか，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．

`srch_tree` 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．

提案した可逆深さ優先探索と実装したプログラムでは，$G = 0$ 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．元の入出力以外のメモリ使用量 M は，提案した可逆深さ優先探索では局所的に定義した変数のみの $\Theta(1)$ だが，実装したプログラムでは $\mathrm{O}(n)$ である．実行時間は常に根の要素で停止するので，$\Theta(n)$ である．よって，時間計算量では一般解法の $\mathrm{O}(n)$ に劣る．

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

図 8 再帰関数を用いた木構造の可逆深さ優先探索

5 評価

RQ1 の評価は，手動で設計した可逆線形探索の元の入出力以外のメモリ使用量 M が $\mathrm{O}(1)$ であったことから Bennett 法で作成した可逆線形探索より優れているものができたといえる．また，入出力の違いが可逆線形探索のごみ出力量に影響を及ぼすことを示すことができた．

RQ2 の評価は，可逆線形探索では探索の成否によって入力ファイルの走査回数に変化が生じた．出力結果によって走査回数に変化が生じることを示すことができた．

RQ3 の評価は，木構造の可逆深さ優先探索を設計するための課題を発見した．また，その課題を解決するための設計方法を提案した．実装は今後の課題とする．

6 考察

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可逆アルゴリズムでは，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設計を行う必要があるといえる．

木構造の可逆深さ優先探索では，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．これは，木構造の可逆深さ優先探索を利用する際の選択肢を増やすことができたといえる．

参考文献

[1] Axelsen, H.B. and Yokoyama, T.: Programming Techniques for Reversible Comparison Sorts, *APLAS*, Feng, X. and Park, S. (Eds.), Vol.9458, pp.407–426, Springer-Verlag (2015).

[2] Bennett, C.H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol.17, No.6, pp.525–532 (1973).

[3] Frank, M.P.: Reversibility for Efficient Computing, PhD Thesis, Massachusetts Institute of Tech-nology, pp.239–243 (1999).

[4] 森田憲一：可逆計算，pp.2–6，近代科学社（2012）．