可逆線形探索の解析

2016SE082 外川淳平　2016SE085 鳥居大樹　2016SE098 吉田翔亮
指導教員：横山哲郎

1　はじめに

本研究では，既存の可逆線形探索の解析方法の改善と解析結果の変化を比較し，提案した解析方法がよりよい方法なのかを検証する．加えて, 同じ探索アルゴリズムである深さ優先探索の可逆化を行う．計算過程においての可逆とは，直前の状態が高々一意に定まるもので，可逆な場合，状態から状態への遷移が 1 : 1 の関係になっている．アルゴリズムは計算機上で問題を解くための手法である．アルゴリズムには線形探索アルゴリズムやバブルソートをはじめとした多くのアルゴリズムが存在し．その中に連結無向グラフの深さ優先探索や幅優先探索が存在する．実際にプログラムを記述して実行させると計算過程で情報を失うことによって，エネルギーを消費し熱を発生させる．一般に非可逆計算を行うと必ず熱が発生してしまうことが分かっている．反対に，可逆計算では計算過程で情報を失わないのでエネルギー消費や発生する熱の下限が存在しない．非可逆なアルゴリズムを可逆化させ，可逆アルゴリズムにする．すなわち，各実行状態に移る計算が単射であり，直前の状態が高々一意に定まるプログラムで記述すると非可逆なプログラムで本来発生するはずだったエネルギー消費や熱が抑えられる可能性がある．基本的な探索アルゴリズムの 1 つである線形探索では，一般解法を用いた可逆化が行われており, さらに手動で可逆化することで一般解法よりも時間計算量の効率の良いアルゴリズムも提案されている [2]．また，深さ優先探索や様々な比較ソートでも可逆化が行われている [1][3]．しかし，深さ優先探索の可逆化は手動による効率化が不十分であると考えられる．そのため，効率のいい手動の解法の提案が求められている．本研究では，はじめに可逆線形探索の既存の解析方法の改善を行う．文献 [2] では，解析が行ごとに行われている．しかし，計算量に多く関わる重要な演算を決めその演算の計算量を求めるほうが解析効率が良いと考えられる．解析方法の変更による結果の変化を明確にする．また，深さ優先探索の C 言語での実装と可逆化を行う．解析方法の改善は，はじめに文献 [2] から解析されたプログラムの結果を用意する．次に行ごとに解析された結果と重要な演算の部分を決め，その部分だけの計算量を求める方法の結果を比較し，結果を記述する．

2　関連研究

2.1　線形探索

線形探索はリニアサーチとも呼ばれ, 数ある探索の中で，基礎的な探索方法である．また線形探索には，様々な方法があり，ここでは 3 つの方法を示す．はじめに，通常の線形探索の方法を示す．通常の線形探索では探したいキーの値と配列の先頭の値が一致するか確認し，一致すれば終了し，一致しなければ，確認した値が配列の最後であるかを確認し，最後の値でなければ配列の次の値を確認し，キーの値と一致するかもしくは配列の最後の値を探索するまでこの探索をつづける．探索成功の場合 1，失敗の場合 0 を返し成否を判定する．次に番兵法と呼ばれる方法がある．番兵法は配列の最後にキーと同じ値を加え，通常の線形探索と同じ方法で探索を行い，キーと同じ値を見つけたときに配列の最後かどうかの確認を行い，配列の最後ならば失敗とし, そうでないならば成功とする．通常の線形探索と比べて，キーと同じ値を見つけたときにだけ，配列の最後かどうかの確認を行えばよいので，比較回数を減らすことができる．最後に整列リストを用いた線形探索と呼ばれる方法がある．整列リストを用いた線形探索はあらかじめ配列の値が昇順または降順にソートされている場合に用いることができる．昇順の場合で考えるとする．まず番兵法のように配列の最後に∞を加え，先頭から探索を行い，探索している配列の値がキーよりも小さい値ならば次の値を探索し，キー以上の値だった場合キーと一致するかどうかを確認し，一致した場合成功，しなかった場合は失敗とする．

2.2　グラフ

グラフは隣接リストの集合による表現と隣接行列による表現の 2 種類の表現方法がある．本研究では隣接リストによる表現を用いる．隣接リストによる表現は $G = (E, V)$ で表現される．V は頂点の集合 E は頂点間を結ぶエッジの集合を表す．グラフには，エッジが有向と無向の 2 種類がある．また，エッジに重みを追加することで，最短経路を求めるアルゴリズムに用いられる．このようにグラフを用いることによって，様々なアルゴリズムを表現することができ，本研究で取り扱う深さ優先探索や幅優先探索にも用いられている．本研究では，無向グラフ表現によるグラフアルゴリズムを取り扱う．

2.3　深さ優先探索

深さ優先探索は縦型探索とも呼ばれ，始点からの深さが大きい未探索の頂点を優先的に探索するアルゴリズムである．手順は，はじめに，与えられた始点と隣接する頂点を探索済みにし隣接する頂点が全て探索済みになるまで行う．隣接するの頂点が全て探索済みの場合，ひとつ前の頂点に戻る．これを繰り返し，全ての頂点が探索済みになるか，探索したい頂点が見つかったときに探索が終了する．始点からの深さが無限にあるグラフでは無限の深さまで探索してしまうため，解が必ず見つけられるとは限らない．また始点からの深さが大きい頂点を優先的に探索する性質

があるため，解が複数個ある場合, 最小経路が最初に見つかるとは限らない.

2.4 一般解法

可逆化を行うための一般解法には幾つかの方法が存在し，その中で Landauer 法と Bennett 法と呼ばれるものがある [4]．Landauer 法は，代入などの非可逆な計算で情報の消失が起こる．そこで非可逆な計算が行われる前に，その計算によって消失する情報を別の場所に保存することで可逆化を実現した．しかしこの方法は元の非可逆な計算の失われる情報を保存するため，実行時間に比例したメモリ使用量が必要となる．Landauer 法の欠点として出力が必要以上に出てしまうということが挙げられる．Bennett 法は Landauer 法の欠点を解消する解法で，プログラムを実行した後に出力に有効な情報を保存し，その後逆実行を行う．逆実行を行うことでスタックを空にし，メモリ使用量を抑えることができる．しかし，逆実行を行うため, 実行時間が約 2 倍になってしまうという欠点がある．また，Lange-McKenzie-Tapp 法と呼ばれるものもあり，チューリング機械を可逆的にシミュレートする方法で，他の計算モデルにおいても一般化できる．この方法のメリットは，構成サイズがある制限の値を超えると探索を中断するので，空間計算量の値を少なくできる．また最悪の場合の漸近的な実行時間が増えない点もある.

2.5 Janus

Janus は命令型可逆プログラミング言語であり，Janus で書かれたプログラムは可逆性が保証されている．非可逆な言語において条件分岐文や繰り返し文は可逆性を持たない．しかし，Janus ではこれらの文も可逆性が保証されている．Janus では，条件分岐文や繰り返し文に可逆性を持たせるために，アサーションと呼ばれる操作を条件分岐文や繰り返し文のあとに加える．アサーションを加えることで, 条件分岐文や繰り返し文で起こる情報消失をなくすことができる. アサーションの操作に入るには，条件分岐で真だった場合, アサーションの条件が真でなければならない. 同様に偽の場合，偽でなければならない.

3 アルゴリズムの解析方法の改善

3.1 アルゴリズムの解析

実際にアルゴリズムをプログラムで実装するとなると，同じアルゴリズムでも実装の方法によってプログラムの動き方が変わってしまう．そのため，プログラムをある指標を用いて数値に表し，同じ指標を用いて数値に表した別のプログラムと比較することでそれぞれのプログラムの良さを把握することができる．このプログラムの良さを見積もることをアルゴリズムの解析と呼ぶ．文献 [2][1] ではプログラムの良さを表すための指標として時間計算量，空間計算量，ゴミ出力量の 3 つを用いているが，本研究では時間計算量に着目する.

3.2 時間計算量

時間計算量はアルゴリズムを実行した際にどの程度時間がかかるかを表す．しかし，アルゴリズムの実行時間はプログラムや実装する計算機，入力されたデータの量によって変化する．そのため，一般的には O 記法という考え方を用いてアルゴリズムそのものの効率を漸近的に考える．具体的にはそのアルゴリズムにとって最も重要とされる演算を基本演算と呼び，基本演算が実行される回数を求めるだけで良いとされる．[5]．アルゴリズム同士では O を用いて比較することが望ましいが，同じアルゴリズムを別々のプログラムで実装した場合だと，大抵は同じ結果が出てきてしまう．文献 [2] の可逆プログラムは全ての行を同一の時間で計算するものとし，行毎に実行回数を評価して時間計算量を求める方法を用いている.

3.3 効率化に出る特徴

効率化を行うとき，一般的に時間計算量を削減すると空間計算量が増加し，空間計算量を削減すると時間計算量が増加するというトレードオフという性質がある．文献 [2] ではそのトレードオフ性が見られず両方とも削減できるか，一方だけ削減しもう一方は変化しないというような結果が出ている．トレードオフ性が出てこなかった原因として 2 つ考えられた．1 つは，一般解法による可逆化は時間計算量と空間計算量のどちらにおいても非常に効率の悪いものとなっている可能性がある．もう 1 つは，計算量の求め方が悪い可能性がある．時間計算量で全ての行を走査したが，そのアルゴリズムにおいてあまり重要でない部分も走査してしまっていた．また，キーを見つけても最後まで走査するという条件に制限することで効率化を行ったものがあり，より狭い条件でのみ効率化が行えることもあるということがわかった.

3.4 基本演算の設定

漸近的な時間計算量を求めるためには，そのアルゴリズムにおいてもっとも時間計算量を支配する演算のみを計算する方法が有効である．ここでは時間計算量を支配する演算を基本演算とする．基本演算を選択し, その部分のみの時間計算量を求める方法は，複雑なアルゴリズムになればなるほど有用性が増す. アルゴリズムが複雑な場合，全ての操作の時間計算量を求めると膨大な時間がかかってしまうが, 基本演算のみの時間計算量を求める場合，O(1) となる操作は計算しないため，求める時間を減少させることができる. 本研究では, 解析を行う際に，この基本演算に着目する方法を用いた.

3.5 別の方法での解析

3.5 節を踏まえて, 時間計算量の基本演算を制限しつつ，解析を行い直した．今回の基本演算は，探索している要素が探索したいものと一致するかを比較する演算，探索している要素が最後の要素かを比較する演算と定義し，それぞ

れ key，index と呼ぶことにした．具体的なプログラム上でどの部分を指すのかは図 1 に示す．また比較するため

```
1  procedure linear1(int k[], int n, int key, int
       f, stack g)
2    local int i = 0
3      push(1, g)
4      from top(g) = 1 loop
5    if k[i] = key then     /*k [i] = key を基本演算
         key とする*/
6        push(f, g)
7        f ^= 1
8        push(1, g)
9      else
10       push(0, g)
11     fi top(g) = 1
12     i += 1
13     push(0, g)
14   until i >= n || f != 0     /*i >= n を基本演算
         index とする*/
15   push(i, g)
16   delocal int i = 0
```

に，文献 [2] にある通常の線形探索 1 の一般解法と提案解法，番兵法の一般解法，のプログラムを用いる．また探索する値が存在する要素の添字を変数 M，探索が成功したら 1 失敗したら 0 を表す変数 S を用意する．ただし，変数 M は探索が失敗したときに全ての要素の数を表す変数 n となる．以上の条件で解析を行った結果を表 1 に示す．解析に

表 1　基本演算別に見た時間計算量

基本演算	全ての行	key	index
通常の一般解法	$14M+18S+22$	$2M+2S$	$2M+2S+2$
通常の提案解法	$5M+6S+6$	$M+S$	$M+S$
番兵法の一般解法	$8M+4S+32$	$2M+2$	2
番兵法の提案解法	$3M+2S+8$	$M+1$	1

M を用いたが,M は探索する値によって変化し，結果に大きく関わる．また，S は探索が成功か失敗かで時間計算量が微量に変化するため, この M，S を用いて時間計算量を表す．通常の線形探索の提案解法では, 一般解法と比較すると,key では $M+S$,index では $M+S+2$ 改善されている．全ての行を計算した場合,$9M+12S+16$ 改善されているが,key と index の部分, つまりアルゴリズムにおける重要な演算の計算量は, 約 $2M$ しか改善されていない．番兵法の行を参照する．番兵法は探索した配列が最後かどうかの比較の回数が通常の線形探索に比べて,index の回数が抑えられていることがわかる．このことから番兵法の解析結果は妥当であると考える．アルゴリズムの重要な部分のみを計算し，類似するアルゴリズムと比較することで，アルゴリズムの性質を解析結果から読み取ることもできる．また, 重要な演算のみ数える方法は, 可逆化前のプログラムのプログラムと計算量がほとんど変化がない. しかし全ての行を数えると可逆化後のほうが計算量が増加する. これは可逆化を行うときに, アサーションを置く必要があるためである．アサーションの条件を変更することで, 全ての行を数えたときの解析結果の計算量は減少するが, 重要な演算のみ数えた場合, 計算量は減少しない．一般解法で可逆化されたプログラムのアルゴリズム部分の改善がされている．Bennett 法による影響で一般解法のプログラムの全体の時間計算量が大体 2 倍となっている．今回の解析で通常の線形探索も番兵法も大体 2 分の 1 の量となっていることから，可逆化によってアルゴリズム本来の動きも変化して，計算量が増加している可能性がある．一方で，可逆プログラムでは可逆制約によって時間計算量が増加するなど，非可逆プログラムには無かった部分の改善も考えなければならない．今回の解析方法ではアルゴリズム部分しか見ておらず，可逆制約による計算量等の解析を行うことができない．しかし，今回の解析方法を応用して，既存の解析方法における各行がどのような役割を持つか場合分けをして解析をする方法を取れば各役割における計算量が分かりやすくなる可能性がある．

4　深さ優先探索の実装と可逆化

4.1　C 言語での深さ優先探索の実装

C 言語で行列と再帰を用いた深さ優先探索を実装し, 以下の図 2 にプログラムを示す．このプログラムを用いて次

```
1  void dfs1(int i, int (*G)[10], int *visited,
       int n)
2  {
3      int j;
4      visited[i] = 1;
5
6      for (j = 0; j < n; j++)
7          if (!visited[j] && G[i][j] == 1)
8              dfs1(j, G, visited, n);
9  }
```

図 1　行列と再帰を用いた深さ優先探索

節で可逆化を行う.

4.2　可逆化の実装

可逆線形探索の文献 [2] の変換規則を元に図 2 の可逆化を行った. 可逆化したプログラムを以下の図 3 に示す．一般解法を用いた可逆化を行った時，if 文，for 文を用いたときに必ずスタックに push してしまうため，メモリ使用量が増えやすくなる．また，変換規則には書かれていないが C 言語でスタックを用いていた場合に Janus と push と pop の操作が少し違うので local 変数を経由して操作をする必要があることに注意した．探索アルゴリズムにおいて一直線に並んだデータ構造の場合，単方向リストでは特定の要素へアクセスするのに $O(n)$ の空間計算量を必要とするが双方向リストでは $O(1)$ でアクセスできる．左方向から右方向へと遷移すると考えると，単方向リストの時は左側の要素がへ戻ることができないため，特定の要素までの全ての要素を記憶する必要があるが，双方向リストでは添字のみを記憶すれば良いからである．木の場合，単方向リストでは同様に $O(n)$ の空間計算量を必要とし，双方

```
1  procedure dfs(int i, int g[][], int vis[],
2  int n, int key, int f, stack s, stack t)
3   local int j = 0
4   local int temp = 0
5    vis[i] += 1
6    temp += i
7    push(temp, s)
8    if i = key then
9      f += 1
10     temp += 1
11     push(temp, t)
12   else
13     push(temp, t)
14   fi top(t) = 1
15   temp += 1
16   push(temp, t)
17   from top(t) = 1 loop
18     if f = 0 && vis[j] = 0 &&
19     g[i][j] = 1 then
20       call dfs(j, g, vis, n, key, f, s, t)
21        temp += 1
22       push(temp, t)
23     else
24       push(temp, t)
25     fi top(t) = 1
26     j += 1
27     push(temp, t)
28   until j >= n
29   if f = 0 then
30     temp += 1
31     push(temp, t)
32   else
33     push(temp, t)
34   fi top(t) = 1
35   push(j, t)
36 delocal int temp = 0
37 delocal int j = 0
```

図 2 可逆深さ優先探索の一般解法

向リストで表現したものを用いるならば O(1) の空間計算量を必要とする．また，木構造は一直線のデータ構造の各要素の子となる要素の数を一つを拡張したものであり，更に閉路を含むように拡張したものがグラフといえる．拡張されたものであるといえるならば，グラフも同様の空間計算量を必要とするだろうと言うことができるはずである．Landauer 法を用いた可逆アルゴリズムは単方向のように特定の動きまでの全ての動きを記憶しておくもので，特定の情報だけで全ての動きが特定できる可逆アルゴリズムを考えることができればより効率のいいアルゴリズムとなるはずである．また，一直線の場合と違い木の場合では次に遷移する場所を特定するために直前の要素を記憶する必要があり，アクセスした要素の添字と合わせて 2 つのデータを記憶する必要がある．このように同じ O(1) でも保持しているデータ量は異なることがある．以上のことを踏まえると，出力の解釈の仕方を工夫するのが効率化において重要なことだと考えられる．具体的には分岐を制御するスタックを減らすために，深さ優先探索の順路を記録するスタックの pop をなくしてどの順番で辿ったのかの判断材料として利用するといったものである．

4.3 実装上の問題

実装の際の問題点として, 文献 [2] の変換規則ではスタックが対応されてなく，新たに考える必要があった．Janus では代入を用いることができないため，push と pop の際に代入の代わりに代入するデータと代入される要素を入れ替えることによってスタックが実現されている．C 言語で実装したスタックで代入を用いていた場合，代入しようとしている変数の値が C 言語では変化しないが，Janus では 0 という値に変化してしまう．ここで, この問題が Janus における問題か, 実装の仕方の問題なのか, 考える必要がある. この問題は C 言語でスタックの push と pop を実装する際に代入を行わず Janus と同様に入れ替えを用いれば，Janus と C 言語で同じ動きになるため，変換規則が必要なくなるため，実装の問題と考えられる．今回は Janus プログラムで局所変数に push や pop を行う値を保存させて push や pop を行うことで問題を解決した．変換規則に局所変数に push や pop を行う操作加えることで, 変換規則をそのまま使用しても実装できる．

5 おわりに

本研究では，可逆線形探索アルゴリズムの解析方法について議論し，改善方法を示した．既存の解析方法では，計算量を求める場合に，アルゴリズムに影響が少ない部分も数えていたが，重要な演算を決定しその部分だけの計算量を比較することで，プログラムの改善によるアルゴリズムの効率の変化がわかりやすくなった．また重要な演算を決定する方法では，そのアルゴリズムの特徴が現れることがあり，アルゴリズムの効率化を行う際に有用となると考えられる．また深さ優先探索の可逆化を行い，Janus での設計を行った．本研究では，行列を用いて深さ優先探索を実装した．今後の課題としては，深さ優先探索の可逆化は行ったが, 効率化が行えていないので，プログラムの改善が課題と考える．

参考文献

[1] 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 (2019).

[2] 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).

[3] Axelsen, H.B. and Yokoyama, T.: Programming Techniques for Reversible Comparison Sorts, *Proc. Programming Languages and Systems* (*APLAS 2015*), Feng, X. and Park, S. (Eds.), Lecture Notes in Computer Science, Vol.9458, pp.407–426 (2015).

[4] Frank, M.P.: Reversibility for Efficient Computing, PhD Thesis, MIT (1999).

[5] 大堀 淳，Garrigue, J.，西村 進：コンピュータサイエンス入門：アルゴリズムとプログラミング言語，pp.3–95，岩波書店 (1999).