卒　業　論　文

可逆グラフアルゴリズム

— 副題ああだこうだ —

2016SE085　**鳥居 大樹**
2016SE098　**吉田 翔亮**

指導教員　**横山 哲郎**

2020 年 1 月

南山大学 理工学部 ソフトウェア工学科

Title Foo and Bar

— Subtitle Bar or Foo —

20yyMTxxx　NANZAN Taro
20yyMTxxx　NANZAN Hanako

Supervisor　SIDO Kyoju

Month 2020

Department of Software Engineering
Faculty of Science and Engineering
Nanzan University

要約

ここに本論文の要約を書く。要約は論文のエッセンスを抜き出したものであるので、ここで取り扱う問題、その問題を解決するための手法、および主な成果が書かれていなければならない。要約を読むだけで、論文の概要が分かり、読者にとって興味を抱く内容か否かが分かるようになっている必要がある。

日本語と英語の両言語で要約を書くが、必ずしも 1 対 1 に対応する文章になっている必要はない。それぞれにふさわしい表現があるからである。

Abstract

In this part, the abstract of this paper is described. Since 'abstract' means the essence or summary of the paper, the abstract should include the description on the problems treated in the paper, the author's approach for solving those problems, and the main results. Readers will understand the outline of the paper, without reading other parts, and be able to decide whether they will have interests in the paper or not.

目次

第 1 章　はじめに 1
- 1.1 背景 . . . 1
- 1.2 目的 . . . 1
- 1.3 アプローチ . . . 1
- 1.4 役割分担 . . . 2

第 2 章　関連研究 3
- 2.1 探索アルゴリズム . . . 3
- 2.2 線形探索 . . . 3
- 2.3 グラフ . . . 4
- 2.4 グラフアルゴリズム . . . 4
- 2.5 深さ優先探索 . . . 5
- 2.6 幅優先探索 . . . 5
- 2.7 一般解法 . . . 5
- 2.8 Janus . . . 6
- 2.9 可逆フローチャート . . . 6
- 2.10 O 記法 . . . 6

第 3 章　グラフアルゴリズムと可逆化 7
- 3.1 アルゴリズムの解析 . . . 7
- 3.2 時間計算量 . . . 7
- 3.3 空間計算量 . . . 7
- 3.4 ゴミ情報 . . . 7
- 3.5 アルゴリズムの解析方法 . . . 8
- 3.6 既存の可逆線形探索 . . . 8
- 3.7 既存の解析方法と問題点 . . . 10

第 4 章　おわりに 13

参考文献 14

付録 A　プログラムリスト 15

付録 B　実行例 50

第 1 章

はじめに

本研究では，可逆アルゴリズムの効率化の整備が進んでいないグラフアルゴリズムの可逆化および効率化を目標とする．また，本研究では，グラフアルゴリズムの中でも基本的なアルゴリズムである深さ優先探索と幅優先探索を取り扱う．

1.1　背景

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．これは非可逆計算を行うと必ず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の消費や熱を発生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の消費や熱の発生が抑えられる可能性がある．また，基本的な探索アルゴリズムの一つである線形探索では，提唱されている一般解法を用いた可逆化が行われている．加えて，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．加えて，似たケースで使われる幅優先探索の可逆化も求められている．

1.2　目的

本研究では，はじめにすでに解析された可逆線形探索とか逆深さ優先探索の解析方法の改善を行う．参考文献 [1][2] では，解析を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の効率が良いと考えられる．解析の方法の変更による結果の変化を明確にする．また，連結無向グラフの深さ優先探索の既に可逆化された一般解法と比較してより効率的な解法を提案すること，連結無向グラフの幅優先探索の一般解法による可逆化と手動による効率的な解法を提案することをを目標とする．可逆化したプログラムと効率化したプログラムの解析を行い，トレードオフの関係を明確にする．

1.3　アプローチ

はじめに，C 言語のプログラムを用意し，一般解法を用いて可逆化を行う．そして可逆化されたプログラムの解析を行い，時間計算量がどの程度かかるのかを把握する．その後可逆化されたプログラムの改良を行い時

間計算量の効率化を図る．最後に改良したプログラムの解析を行い，どの程度効率化できたかを明確化する

1.4　役割分担

後で記述

第2章

関連研究

2.1　探索アルゴリズム

探索アルゴリズムとは，入力されたデータの集合の中から目的となる値や状態を見つけるアルゴリズムである．探索アルゴリズムの分類としてリスト探索，木探索・グラフ探索，文字列探索といったものが挙げられ，本研究で扱う深さ優先探索及び幅優先探索は木探索・グラフ探索の分類に属する．また木探索・グラフ探索の分類には他にも，双方向探索や反復深化深さ優先探索等が挙げられ，その中で深さ優先探索と幅優先探索は基本的なアルゴリズムといえる．

2.2　線形探索

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．通常の線形探索の他に番兵や，整列リストを用いた線形探索がある．番兵法は配列の最後にキーと同じ値を加え，通常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に加えた値かどうかの確認を行い，加えた値ならば失敗とし，加えた値でないならば，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．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．

2.2.1　可逆線形探索のフローチャート

文献 [2] のプログラムを元に可逆線形探索の通常の場合のフローチャートを以下の図に示す．
このプログラムは loop 文の中に if 文は入っている．まずはじめに i を初期化し，$top(g) = 1$ というアサーションを満たすために，$push(1, g)$ を実行する．loop 文のアサーションは繰り返しの実行前は真，繰り返しの実行中は偽でなければならないという性質があるため，$push(1, g)$ を行っている．次に制御文 $i > n - 1 || f! = 0$ の真偽を判定し，真ならば $push(i, g)$ を実行して終了する．偽の場合，if 文の制御文 $k[i] = key$ の真偽を判定する．どちらかの処理が終了すると if 文のアサーション $top(g) = 1$ があり，制御文が真だった場合，このアサーションも真でなければならない．制御文が偽の場合も同様で，このアサーションも偽でなければならない．このアサーションを満たすために，真の場合 $push(1, g)$，偽の場合 $push(0, g)$ を実行している．次に i の値を 1 増やし，また loop 文のアサーションを満たすために $push(0, g)$ を実行する．これを制御文 $i > n - 1 || f! = 0$ が偽になるまで繰り返す．このプログラムの出力は i となるが，$i > n$ の場合，探索している値が見つからな

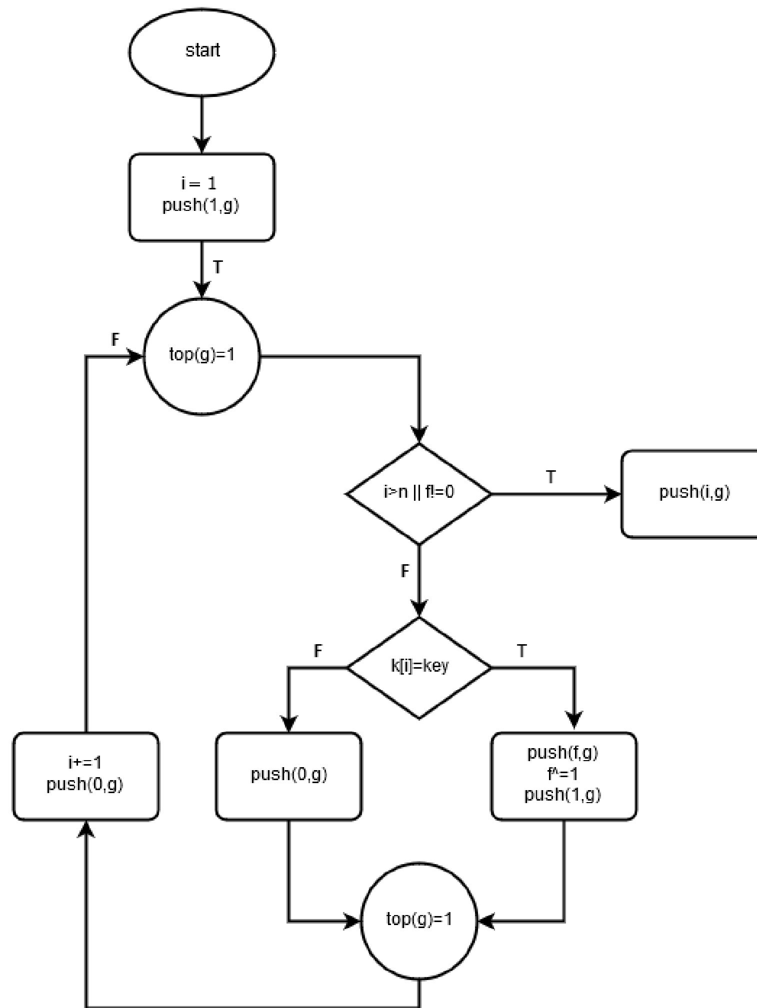

図 2.1　可逆線形探索のフローチャート

かったことを意味している．

2.3　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．

2.4　グラフアルゴリズム

グラフアルゴリズムには様々な探索方法が存在し，探索方法を分類すると，深さ優先探索，幅優先探索，双方向探索，分枝限定法の 4 つの基本的な探索方法に分類される．深さ優先探索を応用したものには，深さ制限探索，反復深化深さ優先探索がある．幅優先探索を応用したものには，最良優先探索，均一コスト探索，$A*$ がある．基本的な探索方法である深さ優先探索と幅優先探索の効率化できれば同様のアプローチもしくは，それ

に近い方法で応用した探索は効率化できる可能性が高い．そのため深さ優先探索と幅優先探索は，グラフアルゴリズムの中でも基本的なアルゴリズムだといえる．本研究では，グラフアルゴリズムの中の基本的なアルゴリズムの効率化を行うことによって，応用的なグラフアルゴリズムの効率化にも貢献できると考える．

2.5　深さ優先探索

深さ優先探索は縦型探索とも呼ばれ，始点からの距離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すべて探索済みになるまで行う．隣接済みの頂点がすべて探索済みの場合，ひとつ前の頂点に戻る．これを繰り返し，すべ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．始点からの距離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．また始点からの距離が大きい頂点を優先的に探索する性質があるため，解が複数個ある時最小経路が最初に見つかるとは限らない。

2.5.1　深さ優先探索の具体的な手順 [2]

D1. 最初に始点となる頂点を選択する.
D2. その頂点を探索済にする.
D3. 隣接する頂点が未探索であれば D4 を実行し, 全て探索済であれば D5 を実行する.
D4. 頂点を探索して探索済として, 探したい頂点とその頂点が一致するか確認し, 一致しない場合は再び D3 を実行する.
D5. 全ての頂点を探索していなければ,1 つ前の頂点に戻り,D3 を実行する.

2.6　幅優先探索

深さ優先探索とは反対に横型探索とも呼ばれ，始点から距離が小さい頂点を優先的に探索する．手順は，始点からの距離が小さい頂点を順に探索する．これを繰り返し，すべ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．深さ優先探索とは反対に始点からの距離が小さい頂点を優先的に探索する性質があるため，解が複数個ある時最小経路が最初に見つかる．

2.7　一般解法

可逆化を行う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は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非可逆な計算によって消失する情報を別の値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，逆実行を行うことによって実行時間が約 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．

2.8　Janus

命令型可逆プログラミング言語 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．本研究では可逆なプログラムを作成する際に，Janus で実装する．

2.9　可逆フローチャート

可逆フローチャートは，Janus などの可逆プログラミング言語をフローチャートで表現するときに用いられる．Janus の loop 文の可逆フローチャートを以下の図 2 に示す．

これは from $e1$ do $S1$ loop $S2$ until $e2$ という loop 文の場合の図である．$e2$ は loop 文の制御文にあたりこ

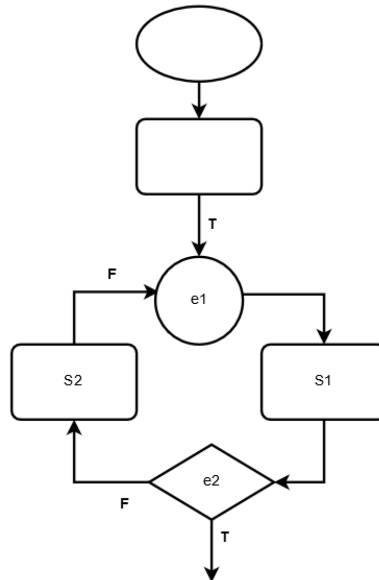

図 2.2　loop 文の可逆フローチャート

の制御文が真ならば loop を抜けることができる．$e1$ はアサーションとしておかれ，繰り返しの実行前は真，繰り返しの実行中は偽でなければならない．loop 文の流れとしては，まずアサーションを満たすための処理が行われ，loop 文の繰り返しが始まる．最初に処理として $S1$ が行われる．$S1$ の処理が行われたあと，制御文 $e2$ の真偽を判定し，真ならば loop 文を抜け出し繰り返しが終了する．偽の場合，$S2$ の処理が行われる．ここでは，アサーションが偽であることが必要である．これを制御文が真になるまで繰り返す．

2.10　O 記法

計算量を漸近的に表す記法で O で表される．$f(x) = 3x + 2 (x \to \infty)$ で計算量を表せるプログラムがあるとすると，O(x) と表す．O は関数 $f(x)$ を実行するのに，x の定数倍の時間程度しかかからず．関数 f は漸近的に x によって上から抑えられることを表す．反対に Ω 記法と呼ばれる $\Omega(x)$ では，漸近的に x によって下から抑えられることを表す．

第 3 章

グラフアルゴリズムと可逆化

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．本研究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，一般的にはオーダー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オーダー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．そのため，本研究では，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．

3.3 空間計算量

空間計算量はプログラムを実行した際にどの程度記憶領域を必要とするかを表す．ここでは変数を 1 つ新たに用意すると記憶領域を 1 使用するものとする．すなわち変数や配列を宣言するか，スタックに push した場合に空間計算量が増える．

3.4 ゴミ情報

ゴミ情報は可逆アルゴリズム特有の考え方で，出力データの中で問題解決に必要な情報を保持している出力データ以外の本来は必要でない出力データをゴミ情報と呼ぶ．非可逆アルゴリズムでは必要な情報のみを出力すれば良いが，可逆アルゴリズムでは逆実行する際の入力として必要な情報も出力される場合があるためゴミ情報と呼ばれるデータが出てくる．また，実行途中で出てくるゴミ情報を中間ゴミ，実行終了時に出てくるゴミ情報を最終ゴミと呼ぶ．

3.5 アルゴリズムの解析方法

本節では可逆線形探索 [] を元にアルゴリズムの解析方法の説明を行う．

3.5.1 線形探索アルゴリズム

線形探索アルゴリズムとは，配列やリスト等のデータの中からあるデータを探し出すための探索アルゴリズムの一つである．データの先頭から順に値を比較し，キーと呼ばれる探索したい値と一致するか，データの末尾まで比較を行うと探索を終了する．入力としては具体的な動きは以下のようになる．

A1．先頭の要素を見る
A2．キーと見ている要素が一致するか比較を行い，一致するなら成功として終了する
A3．次の要素を見る
A4．今見ている要素が存在するなら S2 へ戻り続行する，そうではなくこれ以上探索する要素がないなら失敗として終了する

探索終了した時に探索した要素の場所を出力とする場合と，探索が成功したか否かを出力する場合がある．本稿では後者の探索が成功したか否かを出力するものとして考える．上記が線形探索アルゴリズムの中で最も基本的なアルゴリズムと考えられ，基本的な方法とする，更に拡張したものとして番兵法と整列リストを用いた方法を挙げる．番兵法は A4 の動作を減らすために考えられた方法で，入力データとして用意されたデータの末尾にキーをあらかじめ付け加えるというものである．これによって，例え最初のデータ内にキーがなかったとしても最後に必ずキーを見つけることができ，そのキーの位置が最後か判定することで探索成功か否かを判定する．具体的な動きは以下のようになる

B1．データの末尾にキーを入力する
B2．先頭の要素を見る
B3．キーと見ている要素が一致するか比較を行い，一致したら成功か判定を行い終了する
B4．次の要素を見て，B3 へ戻る

基本的な方法と手順の数は変わらないように見えるが，繰り返し行う手順が基本的な方法だと A2,A3,A4 の 3 つあるのに対して番兵法では B3,B4 の 2 つとなっており，最終的な手順の数では番兵法の方が少なくなっている．しかし番兵法は最初にキーを付け加えるため，データ量が多くなってしまうという欠点もある．次に整数リストを用いた方法はあらかじめデータの値がソートされているものを扱い，昇順ならば比較時に要素がキーの値以上だった場合に終了する．この方法では仮にデータの中にキーがなかったとしてもキーに近い値まで探索を行い探索を終了するため，実行するのにかかる時間が少なくなるというものである．また，データの末尾の次は無限と考えることによって終了条件の手順も繰り返さなくてよくなる．

3.6 既存の可逆線形探索

文献 [] の可逆線形探索では前節で述べられた 3 種類の線形探索アルゴリズムを配列とスタックの 2 通りのデータ構造で可逆化と効率化が行われている．その中でも本稿では通常の線形探索の一般解法と提案解法，番兵法の一般解法の 3 つを取り扱う

プログラム 3.1　通常の線形探索の一般解法 [

```
1 procedure a_linear1_r ( 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
6         push (f, g)
7         f ^= i
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 != n
15    push (i, g)
16  delocal int i = 0
17
18 procedure a_linear_b ( int k[], int n, int key , int f)
19  local stack g = nil
20    local int x = 0
21      call a_linear1_r (k, n, key , x, g)
22      f ^= x
23      uncall a_linear1_r (k, n, key , x, g)
24    delocal int x = 0
25  delocal stack g = nil
```

通常の線形探索プログラムでは key と一致するかの判定を行うたびにスタックへ数値を入れ，そのスタックを出力とすることで，どこまで探索を行ったのかを判別する判断材料にするというものである．また，実行後に逆実行することによって最終ゴミを減らすようにされている．

プログラム 3.2　通常の線形探索の提案解法 [

```
1 procedure linear1 ( int k[], int n, int key , int f)
2   local int i = 0
3     from i = 0 do
4       if k[i] = key then
5         f ^= n
6         f ^= i
7       fi k[i] = key
8     loop
9       i += 1
10    until i >= n -1 || f != n
11    if f = n then
12      i += 1
13    fi f = n
14  delocal int i = f
```

プログラム 3.2 はプログラム 3.1 を改善したプログラムで，スタックを使わないことで空間計算量を減らすように工夫されていた．

プログラム 3.3　番兵を用いた線形探索の提案解法 [

```
1 procedure a_banpei1_r ( int k[], int n, int key , int f, stack g)
2   local int i = 0
```

```
3     push (k[n], g)
4     k[n] ^= key
5     push (1, g)
6     from top(g) = 1 loop
7       i += 1
8       push (0, g)
9     until k[i] = key
10    if i < n then
11      push (f, g)
12      f ^= i
13      push (1, g)
14    else
15      push (0, g)
16    fi top (g) = 1
17    push (i, g)
18  delocal int i = 0
19
20 procedure a_banpei_b ( int k[], int n, int key , int f)
21  local stack g = nil
22    local int x = 0
23      call a_banpei1_r (k, n, key , x, g)
24      f ^= x
25      uncall a_banpei1_r (k, n, key , x, g)
26    delocal int x = 0
27  delocal stack g = nil
```

プログラム 3.3 は先に探索を行い見つかった後に，成功か失敗かの判定を行う．プログラム 3.1 と同様に実行後に逆実行することによって最終ゴミを減らすようにされている．

3.7 既存の解析方法と問題点

文献 [] での時間計算量の解析では，すべての行のステップ数を 2 種類の変数を用いて表していた．プログラム 3.1 を元に具体的な計算方法を以下に示す．

プログラム 3.4 通常の線形探索の一般解法の解析方法 [

```
1  procedure a_linear1_r ( int k[], int n, int key , int f, stack g) \\ 1
2   local int i = 0 \\ 1
3     push (1, g) \\ 1
4     from top (g) = 1 loop \\ M+S+1
5       if k[i] = key then \\ M+S
6         push (f, g) \\ S
7         f ^= i \\ S
8         push (1, g) \\ S
9       else
10        push (0, g) \\ M
11      fi top (g) = 1 \\ M+S
12      i += 1 \\ M+S
13      push (0, g) \\ M+S
14    until i >= n || f != n \\ M+S+1
15    push (i, g) \\ 1
```

```
16   delocal int i = 0 \\ 1
17
18 procedure a_linear_b ( int k[], int n, int key , int f) \\ 1
19   local stack g = nil \\ 1
20     local int x = 0 \\ 1
21       call a_linear1_r (k, n, key , x, g) \\ 1
22       f ^= x \\ 1
23       uncall a_linear1_r (k, n, key , x, g) \\ 1
24     delocal int x = 0 \\ 1
25   delocal stack g = nil \\ 1
```

変数 M は，キーと同じ値がどの位置に存在しているのかを表しており，データの中に存在しない場合はデータの大きさを表す．この M は他の指標と比べて大きな値になりやすく，大きな値になりやすいということはこのアルゴリズムの実行時間に対して与える影響が大きいということである．また，今回の探索アルゴリズムではキーの発見が通常の線形探索で失敗する場合以外での終了条件となっており，実行時間もキーの発見によって影響されやすく，ある程度入力データのキーの位置に法則性があるなら平均的な実行時間を導きだしやすくなっている．したがって，この M という指標は妥当なものだと考えられる．変数 S は，探索が成功したか否かを表しており，通常の場合は探索が成功したか否かで場合分けをして時間計算量を計算しなければならないのを一つにまとめて表すために用いられている．すべての行のステップ数を数えるとされているが，実装時にはプログラムになってしまうため，同じ動きを記述する方法がほんの少し変化するだけで実行時間に大きな影響を与えてしまう場合がある．ここで，アルゴリズムとプログラムの違いについて考えたい．アルゴリズムとは問題解決のための手順を示すもので，プログラムはコンピュータを働かせるために手順に従ってコンピュータへの命令を書いたものである．アルゴリズムの計算量が優れていればそれを元にしたプログラムすべてが優れていると言えるが，プログラムの計算量が優れいてもそのプログラムだけが優れているとしか言えない．よって，効率化を行うならばプログラムよりもアルゴリズムでする方がいい．しかし，すべての行のステップを数えるという方法はプログラムの計算量での比較になってしまう．

3.7.1 別の解析方法

全ての行のステップ数を数えるという指標をキーと比較する演算．探索している要素が最後の要素かを確認する演算の 2 つをそれぞれ数えるという指標にし各演算を key，index と呼ぶ．これらは基本演算とし，アルゴリズムの中で重要な動きをする部分を表している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．

以上の条件で解析を行った結果を表 1 に示す．

表 3.1　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	$14M + 18S + 22$	$2M$	$2M - 2S + 2$
通常の提案解法	$5M + 6S + 6$	$M + S$	$M + S$
番兵法の一般解法	$8M + 4S + 32$	$2M - 2S + 2$	2

通常の線形探索の一般解法と提案解法では key と index のどちらの値も大体 M 改善出来たのに対して key と index 以外の部分では $(14M - 5M) - M - M = 7M$ となり，$7M$ ほど改善できていたことがわかった．この key と index によって改善された合計 $2M$ という値は，アルゴリズムにおいて重要な演算を行う部分であり，プログラムではなくアルゴリズム自体の効率が変化した部分であると考えられる．また，通常の線形探索の一般解法と番兵法の一般解法では key の値はあまり変化がないのに対して index の値が非常に小さくなっているという特徴が見ることができた．これは 3.1 節でも述べた番兵法の特徴であり，こういったアルゴリズムの特徴を確認することができた．基本演算を定めて解析を行うことはアルゴリズムの特徴を発見することが

```
procedure linear1(int k[], int key, int n, int f, stack g)
 local int i = 0
  push(1, g)
  from top(g) = 1 loop
   if k[i] = key then
    push(f, g)
    f ^= 1
    push(1, g)
   else
    push(0, g)
   fi top(g) = 1
   i += 1
   push(0, g)
  until i >= n || f != 0
  push(i, g)
 delocal int i = 0
```

基本演算key

基本演算index

図 3.1　基本演算の説明

でき，アルゴリズム自体の改善にも繋がることが確認できたため必要なことだと考えられる

第 4 章

おわりに

本研究をまとめると、こういうことだったのだ。こうした結果になったのは、この点が良かったと思われる。しかし、あの点については、ああいうことも検討する必要があり、今後の課題として残っている。

謝辞

本研究を進めるにあたり、筆者は何もしてこず、ひとえに周囲の皆様の手とり足とりのご指導によるもので、関係諸氏に深く感謝します。

参考文献

[1] 愛 上男："プログラミング論", 情報処理論文誌, Vol.1 No.1, pp.1-10 (1991.1).
[2] Z.MrX：to Make Programming Much Easier, `http://www.abc.def.gh/ijkl.html`.
[3] 柿 久華子："万能方法論", 日本出版 1990.

付録 A

プログラムリスト

付録 B

実行例

実行例を示す。

```
Hello, world
```