修士学位論文

題目

指向性ランダム探索によるプログラム自動修正

指導教員
楠本 真二 教授

報告者
大田 崇史

平成 27 年 2 月 6 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 26 年度 修士学位論文

指向性ランダム探索によるプログラム自動修正

大田 崇史

内容梗概

ソフトウェア開発における工数の大部分を占めるデバッグに関する工数を削減するために，プログラムの自動修正が望まれる．そのため，近年，遺伝的アルゴリズムを用いたプログラム自動修正手法 GenProg が有望視されている．GenProg は遺伝的アルゴリズムに基づきプログラムの修正を行う．

GenProg は実社会のソフトウェアに存在する故障を修正可能なことを示したが，プログラム自動修正における遺伝的アルゴリズムの有用性は証明していない．これに関して，ランダム探索によりプログラムの自動修正を行う手法 RsRepair は，遺伝的アルゴリズムには計算コストに見合うだけの有用性が無いことを示した．

複数のプログラム記述に起因するソフトウェアの故障に関する修正では，複数のプログラム記述に関する修正が必要となる．しかし，ランダム探索ではある記述を修正した後に新たな故障を導入する可能性が高く，故障の導入なしに複数の記述を修正することは困難である．あるプログラムの故障が複数のプログラム記述に起因することは十分考えられるので，複数のプログラム記述修正にランダム探索を対応させることで，さらなるデバッグの工数削減が期待できる．

本研究ではランダム探索におけるプログラムの自動修正に関する問題を解決するために，テストケースを用いた指向性ランダム探索を提案する．提案手法は変異したプログラムに対するテストケースの適用結果により故障を導入する変異を軽量的に削減する．

本研究では提案手法を実社会のプログラムに適用してその有用性を確かめた．

主な用語

Automated program repair，Directed random search，Genetic programing，Search-based software engineering，

目 次

図 目 次

表 目 次

1　まえがき

ソフトウェアの安全性や信頼性の向上のために徹底的なデバッグが必要である．デバッグとはソフトウェアの故障を取り除く作業のことを示す．一般的に，デバッグはソフトウェアの故障を検出し，ソフトウェアの故障を引き起こす原因となったプログラム中の記述（欠陥）を同定し，同定した欠陥を修正する作業により構成される．

デバッグは工数のかかる作業として知られており，ソフトウェア開発者はソフトウェア開発作業の内，50 % をデバッグに費やすといわれている [1]．また，アメリカにおいて 1 年間にデバッグで費やされる費用は約 3000 億ドルであるという調査もされている [2]．

そのため，デバッグにかかる工数を削減することを目的として，デバッグの自動化に関する研究がなされてきた．ソフトウェアの故障検出はテストケースにより行う．テストケースはソフトウェアに期待する特定の動作を記述しており，ソフトウェアがテストケースを成功することはソフトウェアがそのテストケースに記述した動作を満たすことを示す．逆に，ソフトウェアがテストケースを失敗することはソフトウェアがそのテストケースに記述した動作を満たさないことを示す．ソフトウェア開発者はテストケースの失敗によりソフトウェアの故障を検出する．ソフトウェアの故障検出を徹底的に行うためには，膨大な数のテストケースを生成する必要があり人手で行うと工数がかかる．よって，テストケース生成の工数削減を目的とし，膨大な数のテストケースを自動生成する研究がなされてきた [3–6]．開発者はテストケースによりソフトウェアの故障を検出すると，故障の原因となった欠陥を同定する必要がある．欠陥は膨大なプログラム中の数箇所に存在するため，人手で行うと工数がかかる．よって，欠陥同定に関する工数削減を目的とし，欠陥同定を自動で行う研究がなされてきた [7–9]．デバッグを構成する 3 工程の内，ソフトウェアの故障検出と欠陥同定の自動化については多くの研究がなされており，実用段階であるが，同定した欠陥の自動修正に関する研究はあまり行われていない．さらに，欠陥修正には開発者がプログラムの動作を理解し，どのようにプログラムを修正すれば欠陥が取り除かれるのかを熟慮する必要がある。そのため，欠陥修正は工数がかかる作業となる．つまり，デバッグ作業の完全な自動化にはプログラムの自動修正が必要である．

そこで，近年プログラムの自動修正に注目が集まっている．プログラム自動修正に関する研究は 2 つに大別される．1 つは確率的な手法で，ランダムな工程を繰り返し行うことでプログラムの修正を行う．もう一方はプログラムが満たすべき性質を表す論理式を元に修正を行う．論理式を元にする修正手法では，複雑な論理式を SMT ソルバ　[10] で解く必要がある．そのため，現在の CPU の処理能力では，現実的な時間に論理式を解くことは困難である．よって，本研究では確率的な手法に着目する．

確率的な手法では GenProg が有望視されている [11,12]．GenProg は遺伝的アルゴリズムに基づき修正対象のプログラムに繰り返し変更を加えプログラムの自動修正を行う．GenProg で実社会のソフトウェア中に存在する欠陥を修正可能なことが示されたが，遺伝的アルゴリズムの有用性は示されていない．そこで，Yuhua らは遺伝的なアルゴリズムの必要性を懐疑し，RsRepair を提案した [13]．

RsRepair はランダムにプログラムを変更する作業を繰り返すことでプログラムの自動修正を行う．プログラムの修正にかかる時間やプログラムに加えた変更回数の観点で GenProg より RsRepair が優れており GenProg が用いる遺伝的アルゴリズムが有用でない事が示された．

しかし，RsRepair は修正対象のプログラムを 1 行変更したプログラムしか生成できない．そのため， RsRepair はあるソフトウェアにおける故障の原因が複数行ある場合や，複数の故障を含むプログラムは自動修正できない．また，ランダムな変更を修正対象のプログラムに複数回適用すると，見当違いのプログラムが生成される可能性がある．よって，本研究ではランダムなプログラムの変更を軽量的に制御することで複数行の修正が必要なプログラムを高速に自動修正する手法を提案する．提案手法はプログラムに加えたランダムな変更の妥当性をテストケースを元に判断する．つまり，提案手法は変更前のプログラムが成功するテストを変更後のプログラムが失敗する場合，加えた変更を無効とする．

本研究では，提案手法を実装し複数のオープンソースソフトウェアに適用し，その有用性を示した．

2　関連技術

本節では，本研究に関連する諸技術について述べる．

2.1　指向性ランダム探索

指向性ランダム探索（directed random search）とは，完全にランダムな探索（undirected random search）をなんらかの指標により制御する探索手法である．指向性ランダム探索は，特にテストの自動生成分野において発展しており，テストスイートの網羅率を向上するテストケースを生成する [14]．ここで，完全なランダム探索を制御する指標はテストケースの網羅率やテストの実行可能性である．オブジェクト指向プログラムのテストにおいて実体化していないオブジェクトのメソッドを参照するとプログラムの実行が停止する．このような実行を行うテストの生成をさけるために，完全なランダム探索を制御する．

2.2　遺伝的アルゴリズム

遺伝的アルゴリズムは生物が環境に適応することを工学に応用したアルゴリズムである [15]．遺伝的アルゴリズムはランダムな個体生成及び，個体の評価値に基づく生存個体選択を繰りかえし行うことで動作する．まず，遺伝的アルゴリズムは初期集団と呼ばれる個体群をランダムに生成する．これが第 1 世代となる．個体数は個体群が含む個体数を定義する．生成した各個体の環境適応度は環境を模倣した評価関数により評価される．ここで，環境適応度が高い個体ほど解に近い事を示す．よって，解に近い個体を生存させるために環境適応度に基づき生存個体を選択する．生存個体の選択方法，選択数はアルゴリズム設計毎に異なる．次に，生存した個体を元に次世代を生成する．次生成方法もアルゴリズム設計毎に異なる．遺伝的アルゴリズムは世代生成を繰りかえし行うことで解を探索する．遺伝的アルゴリズムは解の発見または次世代生成回数の規定回数（最大世代数）到達で終了する．

3 既存研究

同定したプログラム中の欠陥を取り除きプログラムを修正することで，ソフトウェアの故障は取り除かれる．プログラム中の欠陥を取り除くには，適切にプログラム中の記述を書き換える必要がある．この作業を行うために開発者はプログラムの動作や記述内容を深く理解し，どのように記述を書き換えるかを熟慮する必要がある．プログラムの動作や記述内容の理解，適切な記述の書き換えを行うのは困難で，工数のかかる作業である．そのため，プログラムの自動修正作業の削減を目的とし，プログラムの自動修正に関する研究がなされてきた．本小節では，プログラムの自動修正に関する諸研究について述べる．

3.1 GenProg

GenProg は遺伝的アルゴリズムに基づきプログラムの自動修正を行う手法である [11,12]．本節では，まず，GenProg の基本的なアイデアを説明するために初期に提案された GenProg [11] について述べる．次に，改良版の GenProg [12] について述べる．

3.1.1 GenProg の概要

GenProg [11] は遺伝的アルゴリズムに基づいてプログラムを自動修正する手法である．GenProg は故障を含むソフトウェア及び少なくとも 1 つの失敗テストケースを含むテストスイートを入力とし，プログラムの自動修正を行う．ここで，入力するソフトウェアを構成するプログラムを修正対象プログラムと呼ぶ．この入力はソフトウェアの故障をテストスイートにより検出することを想定すると自然である．出力は修正プログラムである．修正プログラムは修正が完了したプログラムを示す．また，GenProg はプログラムの自動修正のみを行う手法ではなく，テストスイートをもとに欠陥同定を行い同定した欠陥を修正する手法である．欠陥の同定部分と欠陥の修正部分は独立しているため任意の欠陥同定手法を GenProg に適用することができる．

遺伝的アルゴリズムの設計では個体の表現，個体の操作及び個体の評価関数を設計する必要がある．本小節では， GenProg の動作概要を述べる．個体の表現及び個体の操作及び個体の評価関数の詳細は次小節以降で述べる．

GenProg の入力は故障を含むソフトウェア P 及びテストスイートである．テストスイートは修正対象プログラムが成功するテストケースの集合 $PosT$ 及び修正対象プログラムが失敗するテストケースの集合 $NegT$ の和集合である．ソフトウェアの故障の原因となるものが欠陥である．また，ソフトウェアの故障はテストにより検出する，よって， $NegT$ は空集合ではない．また，GenProg は全テストケースを成功するプログラムを修正プログラムとしている．すなわち，修正プログラムの質はテストケースに依存する．つまり，GenProg により導出した修正プログラムは開発者が望んだ修正を加えたプログラムでない可能性がある．Algorithm 1 に GenProg の擬似コードを示す．まず，1 から 3 行目でテストスイートに基づき欠陥同定を行う．1，2 行目で成功テスト及び失敗テス

Algorithm 1 : The GenProg Algorithm

Input: : Faulty program P

Input: : Testsuite including set of positive/negative testcases $PosT$ and $NegT$

Output: : Patch that repairs the faulty program P

1: $Path_{PostT} \leftarrow \cup_{p \in PosT}$ statements visited by $P(p)$
2: $Path_{NegT} \leftarrow \cup_{p \in NegT}$ statements visited by $P(n)$
3: $C_{sub} \leftarrow$ FaultLozalize$(Path_{NegT}, Path_{PosT})$
4: $Popul \leftarrow$ initial_population$(P, \text{pop_size}, \text{C}_{\text{sub}})$
5: **repeat**
6: $\quad Viable \leftarrow \{\langle P, Path_p, f\rangle \in Popul \mid f > 0\}$
7: $\quad Popul \leftarrow \emptyset$
8: $\quad NewPop \leftarrow \emptyset$
9: $\quad$ **for all** $\langle p_1, p_2\rangle \in$ sample($Viable$, pop_size/2) **do**
10: $\quad\quad \langle c_1, c_2\rangle \leftarrow$ crossover(p_1, p_2)
11: $\quad\quad NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
12: $\quad$ **end for**
13: $\quad$ **for all** $\langle V, Path_V, f_V\rangle \in NewPop$ **do**
14: $\quad\quad Popul \leftarrow Popul \cup \{$mutate$(V, Path_V)\}$
15: $\quad$ **end for**
16: **until** $\exists \langle V, Path_V, f_v\rangle \in Popul.f_V =$ max_fitness
17: **return** minimize$(V, P, PosT, NegT)$

トの適用時に修正対象のプログラムが実行するプログラムの行を導出する．$P(p)$ 及び $P(n)$ はそれぞれ，修正対象のプログラムに対する成功及び失敗テストの適用を示す．$Path_{PosT}$ 及び $Path_{NegT}$ はそれぞれ修正対象のプログラムが成功テスト及び失敗テスト適用時に実行するプログラム中の行を表す．3 行目では $Path_{PosT}$ 及び $Path_{NegT}$ に基づき欠陥同定を行う．欠陥同定ではプログラムの各行に疑惑値を付与する．疑惑値はプログラムの各行における欠陥である可能性を示す値である．4 から 16 行目が GenProg における遺伝的アルゴリズムを示す．4 行目で初期集団の生成を行い，5 から 16 行目の繰り返しで世代生成を繰り返す．世代生成の繰り返しは修正プログラムの発見或いは規定回数の世代生成で終了する．4 行目で関数 initial_population を用いて初期集団を生成する．pop_size は 1 世代の個体数を表す．initial_population は後述する変異処理により修正対象プログラムを変更したプログラムを pop_size 個生成する．6 行目で 1 つのテストケースも成功しない変更プログラムを除外する．次に，各変更プログラムの評価値を評価関数に基づき評価する．このとき，より多くのテストケースを成功する変更プログラムの評価値が高くなる．つまり，より修正プログラムに近いプログラムの評価値が高くなる．評価値に基づき pop_size/2 個の 変更プログラムが次世代に生存する．9 行目は 変更プログラムの評価及び生存個体の選択を表す．生存する変更プログラムの内，2 つの変更プログラム又し次世代の変更プログラムを生成する．$variant_1$ と $variant_2$ を交叉する場合，$variant_1$ の前半部分と $variant_2$ の後半部分で構成される変更プログラム及び $variant_1$ の後半部分と $variant_2$ の前半部分で構成される変更プログラムを生成する．ここで，生存する変更プログラムは交叉に用いられた変更プログラム及び交叉で生成された変更プログラムで構成されることに注意されたい．次に，生存する変更プログラムを突然変異によりさらに変更することで，次世代を生成する．13 から 15 行目が突然変異の処理を示す．遺伝的アルゴリズムは修正プログラムの生成或いは，規定回数の世代生成により終了する．GenProg で用いる遺伝的プログラムの性質上，可読性の低い冗長な修正プログラムを生成する可能性がある．そこで，delta debugging [16] を用いて冗長な記述を削減する，17 行目は修正プログラムの冗長性削減を関数 minimization を用いて行う．次小節以降で，個体の表現，個体の操作，個体の評価関数及び修正プログラムの冗長性削減について詳細に述べる．

3.1.2 個体の表現

GenProg での個体は修正対象プログラムを数行変更したプログラムである．GenProg は各変更プログラムを抽象構文木と重み付きパスで表現する．重み付きパスは抽象構文木の各ノードと各ノードに関する疑惑値で構成するペアの系列である．疑惑値は各ノードが欠陥である可能性を示す．つまり，疑惑値が高ければ高いほどそのノードが欠陥である疑惑が強くなる．各ノードに対する疑惑値の付与は修正対象のプログラムにテストスイートを適用した結果を用いる．Algorithm 1 中の set_weights 関数は各ノードに対する重み付けを行う．GenProg は成功テストが到達するプログラム中の記述は欠陥である可能性が低いとして set_weights 関数を定義する．まず，set_weight 関数は全ノードに疑惑値を 0 付与する．次に，修正対象プログラムに失敗テストを適用した結果到達するノードに疑惑

値 1.0 を付与する．次に，修正対象のプログラムに成功テストを適用した結果到達するノードに疑惑値として W_{path} を付与する．Weimer らは $W_{path} = 0.01$ としている [12]．結果として，失敗テスト及び成功テストの実行共に到達するノードの疑惑値が W_{path} 失敗テストの実行でのみ到達するノードの疑惑値が 1.0，成功テストの実行でのみ到達するノードの疑惑値が W_{path}，テストケースの実行で到達しないノードの疑惑値が 0 となる．

3.2 個体探索空間の限定

GenProg での個体は修正対象プログラムを数行変更したプログラムである．ここで，課題となるのが探索空間である．プログラム中の記述の変更は，変更対象記述の決定及び記述の変更方法の決定から構成する．つまり， 1 行の変更において個体がとりうる空間は変更対象記述数と記述の変更方法数の積となる．さらに，複数行の変更では個体がとりうる空間は変更対象記述数と記述の変更方法数を変更行数でべき乗したものとなる．このように個体がとりうる空間が膨大であるため，現実的な時間でのプログラム自動修正では探索の限定が必要となる．プログラムの修正は欠陥同定及び欠陥修正で構成する．欠陥同定は修正対象プログラムの記述に基づき有限である．しかし，プログラムが膨大になると同定対象も膨大となる．そこで，GenProg は欠陥同定の粒度をプログラムの行単位とする．また，プログラムの記述方法は無限である．すなわち，同定したプログラムの記述をどのように修正するかは無限の探索空間を持つ．そこで，GenProg は ” 既にプログラム中に存在する記述を用いることでプログラム中の欠陥を修正可能である “ という仮定を用いて欠陥の書き換え方法に関する探索空間を既にプログラム中に存在する記述に限定する．この仮定が成り立つ欠陥を含むプログラムの例を図 1 に示す．図 1 のプログラムに期待する動作は，引数として受けとる 2 つの 0 以上の整数 a, b に関する最大公約数の出力及びプログラムの停止である．このプログラムは入力として $a = 0$ が与えられたときに，b の値が 0 以外であったときに期待しない動作を引き起こす．$a = 0$ が与えられると図 1 のプログラムは 2 行目の if 文の条件を満たし，最大公約数として b の値を出力する．次に，プログラムの実行は while 文の条件判定を行う．b が 0 以外のとき，while 文の繰り返し処理を行う．繰り返し処理内では a, b の大小関係に基づき処理が分岐する．$a = 0, b$ が 0 以外のとき a よりも b が大きいので 8 行目の $b - a$ が実行される．$a = 0$ であるため，a, b の大小関係は $a < b$ のままであり，次の繰り返しでも同様の処理が行われる．このように，while 文の繰り返し判定に用いる変数 b の値が変わらないため，このプログラムは $a = 0, b$ が 0 以外の入力のとき停止しない．これは，期待しない動作である． この期待しない動作の原因は 3 行目の直後に return 文が抜けていることである．この文は 14 行目に存在するため，14 行目の文を 4 行目に複製することで図 1 の欠陥を修正できる．図 1 では，プログラム記述の漏れがソフトウェアの故障の原因となっている．このように，プログラム中に記述漏れがある場合，記述漏れがある行の直前の行が欠陥であるとみなすことが多い [17]．つまり，図 1 の例では，3 行目が欠陥となる．GenProg は個体の生成に関する操作を抽象構文木上のノードを操作により行う．GenProg は欠陥同定を set_weights 関数で行うが，GenProg において欠陥同定と遺伝的アルゴリズム部分は独立しているので異なる欠陥同定

```
 1: void gcd(int a, int b){
 2:   if (a == 0) {
 3:     printf("%g¥n", b);
 4:     // return 0; can be repaired by inserting line 18
 5:   }
 6:   while (b != 0) {
 7:     if (a > b) {
 8:       a = a - b;
 9:     } else {
10:       b = b - a;
11:     }
12:   }
13:   printf("%g¥n", a);
14:   return 0;
15: }
```

図 1: 欠陥を含む最大公約数を導出するプログラム

手法を用いることも可能である．

3.2.1 生存個体の選択

GenProg では次世代に生存する個体の選択に SUS（stochastic universal sampling）を用いる [18]．SUS では variant_i の生存確率は各個体の評価値を計算する関数 fitness を用いて式（1）で計算される．

$$prob_i = \frac{fitness(variant_i)}{\sum_{i=1}^{\text{pop\_size}} fitness(variant_i)} \tag{1}$$

Algorithm 1 の 9 行目で用いられる sample 関数は各個体の生存確率に基づき，pop_size/2 個の個体を選択し，$Viable$ に格納する．$Viable$ 中の個体は交叉に用いられた後，突然変異の操作を適用される．

3.2.2 個体の交叉

個体の交叉は 2 つの個体の重み付きパス中に存在する抽象構文木の前半ノードと後半ノードを入れ替える．2 つのプログラムの交叉後は，交叉のもととなるプログラム及び交叉により生成される新しい 2 つのプログラムが次世代に生存する．2 に交叉のアルゴリズムを示す．

3.2.3 個体の変異

GenProg はプログラムの変更で追加，削除及び入れ換えの 3 つの操作を用いる．ある個体に関する変異は 1 箇所について行われる．変異を行う箇所は疑惑値に基づき決定する．

追加操作は変異対象箇所の直後にプログラム中に既存の記述を追加する．
削除操作は変異対象箇所を削除する．
入れ替え操作は変異対象箇所をプログラム中に既存の記述で入れ替える．

Algorithm 2 : The Crossover Operation of GenProg

Input: : Parent variant $Parent_A$ and $Parent_B$

Input: : Paths $Path_A$ and $Path_B$

Output: : Child variant $Child_C$ and $Child_D$

1: $cutoff \leftarrow \text{choose}(|Path_A|)$
2: $Child_C, Path_C \leftarrow \text{copy}(Parent_A, Path_A)$
3: $Child_D, Path_D \leftarrow \text{copy}(Parent_B, Path_B)$
4: **for** $i = 1$ *to* $|Path_A|$ **do**
5: **if** $i > cutoff$ **then**
6: $prob \leftarrow \text{crossover}_\text{p}\text{rob}(Path_A[i])$
7: **end if**
8: **if** rand(0, 1)$\leq prob$ **then**
9: $Path_C[i] \leftarrow Path_B[i]$
10: $Path_D[i] \leftarrow Path_A[i]$
11: **end if**
12: **end for**
13: **return** minimize$(V, P, PosT, NegT)$

3.2.4 GenProg の改良

Algorithm 3 : The Improved GenProg Algorithm

Input: : Faulty program P

Input: : Testsuite T including set of positive/negative testcases $PosT$ and $NegT$

Input: : Mutation operator Mutate

Input: : Crossover operator Crossover

Input: : Full fitness predicate FullFitness

Input: : Sampled fitness SampleFit

Input: : Population size $PopSize$

Output: : Patch that repairs the faulty program P

1: $C_{sub} \leftarrow$ FaultLocalization(P, T)
2: $Popul \leftarrow$ initial_population$(P, \text{pop_size}, \text{C}_{\text{sub}})$
3: **repeat**
4: $\quad Fitnesses \leftarrow$ SampleFit(Popul)
5: $\quad Popul \leftarrow \emptyset$
6: $\quad Parents \leftarrow$ TournSelect$(Popul, PopSize, Fitnesses)$
7: $\quad Offsprings \leftarrow$ CrossOver$(Parents, P)$
8: $\quad Popul \leftarrow$ Mutate$(Parents \cup Offsprings)$
9: $\quad \langle c_1, c_2 \rangle \leftarrow$ crossover(p_1, p_2)
10: $\quad NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
11: **until** $\exists variant \in Popul$. FullFitness$(variant) = Passed$
12: **return** minimize$(V, P, PosT, NegT)$

Goues らは GenProg における変異プログラムの表現方法，生存個体の評価関数と選択及びプログラムの変異方法を改善した [12]．Algorithm 3 に改善した GenProg のアルゴリズムを示す．

個体の表現に関する改良．GenProg はプログラムを表す抽象構文木及び重み付きパスで個体を表現する．そのため，修正対象プログラムの規模が大きくなると 1 世代がメモリ上に収まらない可能性がある．よって，Goues らは各個体を修正プログラムにたいする操作列（パッチ）で表現する．パッチはプログラム中のある記述を別の記述に置き換えるということを指示する．例えば，パッチは add(6, 15)，delete(19) のように修正 z 対象プログラムに対する操作の系列で表される．この例では，修正対象プログラムの 6 行目を 15 行目に追加したあと，19 行目を削除するという操作を示す．

生存個体の評価関数と選択に関する改良．GenProg の処理において最も時間のかかる処理は各 variant に対するテストケースの適用である．テストケースの variant に対する適用は variant の評価値を導出するために行う．variant の評価値は生存個体選択の指標となる．つまり，variant の評価値導出及び生存個体選択改善しテストケースの適用回数を減らすことで，GenProg のスケーラビリ

ティは上昇する．遺伝的なアルゴリズムは，GenProg と同様に各個体に評価値を与える処理が実行時間の大部分を占めることが多い．そこで，評価関数の適用処理を減少する方法が存在する．Goues らは NFF（Noisy Fitness Function） [19] を用いることで，評価関数の適用処理を改善した．NFF の GenProg に対する応用では，入力されたテストスイートの部分集合（サンプルテストスイート）をランダムに生成し，各 variant に適用する．各 variant の評価値はサンプルテストケースの適用結果をもとに決定する．生存個体の選択は tournament selection [20] で行う．Algorithm 3 の 6 行目が NFF 及び tournament selection の処理を示す．修正プログラムは全てのテストケースを成功する．つまり，テストケースを 1 つでも失敗する variant は修正プログラムではない．よって，全てのサンプルテストスイートを成功する variant についてのみ全テストケースを適用し正しさの検証を行う．全テストケースを成功する variant が修正プログラムである．

突然変異に関する改良．Goues らは初期の GenP r og で提案された 3 つの抽象構文木に対する操作 $delete, insert, swap$ の内 $swap$ に効果が無いことを指摘した．そこで，操作の改善案として $replace$ を導入した．$replace$ はプログラム中のある行をプログラム中に存在するべつの記述で置き換える操作である．これは，プログラム中のある行を削除し，削除した行にプログラム中の記述を追加する．つまり，$Replace(i, j)$ はプログラム中のある行 i について $delete(i)$，$add(i, j)$ を連続して行うことを示す．

交叉に関する改良．個体の表現が変わったため，交叉に関する操作も変更する必要がある．個体は修正対象プログラムに対する操作系列で表現する．交叉は 2 つの操作系列を連結し，それぞれの操作系列を半分ずつ残すことで行う．例えば，2 つの操作系列 $P = \{p_1, p_2, p_3, p_4\}$ 及び $Q = \{q_1, q_2, q_3, q_4\}$ を交叉する場合，まず，操作系列の連結 $P\ to\ Q = \{p_1, p_2, p_3, p_4, q_1, q_2, q_3, q_4\}$ と $Q\ to\ P = \{q_1, q_2, q_3, q_4, p_1, p_2, p_3, p_4\}$ を作成する．次に，連結した操作系列中の各操作を 1/2 の確率で削除することで交叉処理を終える．

3.3 RsRepair

Algorithm 4 : The RsRepair Algorithm

Input: : Faqulty program P

Input: : Testsuite T including set of positive/negative testcases $PosT$ and $NegT$

Input: : Mutation operator Mutate

Output: : Patch that repairs the faulty program P

index $\leftarrow 0$
$\{n_0, t_1, t_2, ..., t_n\} \leftarrow T$
$T \leftarrow \{(n_0, 1), (t_1, \text{index}), (t_2, \text{index}), ..., (t_n, \text{index})\}$
$C_{sub} \leftarrow$ FaultLocalization(P, T)
repeat
  $variant \leftarrow$ Mutate(P, C_{sub})
  $Popul \leftarrow \emptyset$
  for $i \leftarrow 0$ to n **do**
    $(t_{index}, \text{index}) \leftarrow$ GetTestCase(T, i)
    if PatchValidation$(P, variant, t_{index}) \neq true$ **then**
      temp $\leftarrow (t_{index}, \text{index} + 1)$
      $T \leftarrow$ Prioritize(T, temp)
      break
    else if $i = n$ **then**
      SuccessFlag $\leftarrow ture$
    else
      continue
    end if
  end for
until SuccessFlag $= true$
return $variant$

GenProg はその有用性を実社会のソフトウェア中に存在する欠陥を自動修正することで 示した．しかし，GenProg の有用性は遺伝的アルゴリズムによるものなのかは証明されていない．実際，GenProg において修正プログラムは第 1 世代中にみつかることが多い．第 1 世代はランダムに修正対象のプログラムを変異することで行われるため，遺伝的アルゴリズムによる制御の影響がない．つまり，GenProg の有用性は遺伝的アルゴリズムによるものなのか，変異操作によるものなかを明らかにする必要がある．遺伝的アルゴリズムは実行時間の大部分を個体の評価と選択に費やす．そのため，遺伝的アルゴリズムが有用となるのは個体の評価と選択に費やすコストに見合うだけ解への収束速度が上昇するときである [21–23].

そこで，Yuhuha らはプログラムの自動修正における遺伝的アルゴリズムの有用性を検証するために，RsRepair を作成した [13]．RsRepair は GenProg と同様の変異操作を用いてプログラムの自動修正を行う．しかし，RsRepair は遺伝的アルゴリズムではなくランダム探索を用いる．そのため，RsRepair は生成した個体の評価を行わない．よって，生成された各個体は正しさの検証のみ行われる．正しく修正を行う個体は全てのテストケースを成功するので，生成した個体が失敗するテストケースを発見するとすぐに処理を打ち切る．したがって，RsRepair において生成する個体は GenProg において生成される個体よりも少ないテストケースが適用される．さらに，個体の正しさの検証における性質上テストケースの優先付け [24,25] が有効となる．つまり，より多くの個体が失敗するであろうテストケースを優先して各個体に適用することで，各個体に適用するテストケースの数を削減する．RsRepair では過去により多くの個体が失敗したテストケースの優先順位を上げることでテストケースの優先付けを行う．

Algorithm 4 に RsRepair のアルゴリズムを示す．RsRepair は欠陥を含むプログラム P と少なくとも 1 つの失敗テストケースを含むテストスイート T を入力とし，欠陥を修正するパッチを出力する．RsRepair はランダム探索により修正対象プログラムを修正するパッチを自動生成する．また，修正パッチの生成過程はテストケースの優先付けにより高速化される．

RsRepair はまず，入力されたテストスイート中の全テストケースに整数（失敗回数）を付与し，テストスイートを再構成する．失敗回数は過去に，どれだけの個体がそのテストケースを失敗したかを示す．この情報はテストケースの優先付けに用いる．各テストケースに付与する初期値は，修正対象のプログラムが失敗するテストケースが 1，成功するテストケースが 0 となる．このテストケースの優先付けは，過去に多くの個体が失敗したテストケースは今後も多く失敗される可能性が高いという直感に基づく．1 から 3 行目がテストケースの優先付けの準備を示す．

GenProg と同様に，修正する箇所を制限するために RsRepair は欠陥同定を行う．欠陥同定は GenProg と同様の手法で行う．

6 から 21 行目が RsRepair における修正プログラム探索の繰り替えし処理である．まず，欠陥同定の情報をもとに修正対象プログラムに対するパッチを個体として生成する．次に，8 から 20 行目で生成した個体をテストケースにより検証する．ここでは，個体が全テストケースを成功するかを検証する．個体があるテストケースを失敗する場合は，そのテストケースに付与する失敗回数をインク

リメントする．このとき，すぐに検証を行う繰り返し処理から抜け次の個体を生成する．10 行目の GetTestcase 関数はテストスイート T から i 番目に優先度の高いテストケースを取り出す．11 行目の PatchValidation 関数は個体があるテストケースを成功するかを判定する．PatchValidation 関数の引数は修正対象のプログラムと個体とテストケースである．つまり，修正対象のプログラムを個体が示す操作で変異し，変異したプログラムがテストケースを成功するかを判定する．11，12 行目は個体があるテストケースを失敗するときの処理を示す．個体があるテストケースを失敗するときそのテストケースの失敗回数をインクリメントする．そのあと，Proptize 関数によりテストケースの優先度付けを再度行う．次の個体の生成では，修正対象のプログラムの変異により生成する．つまり，1 つ前に生成した個体の変異は行わない．修正プログラム探索の繰り返しは，全テストケースを成功する個体が見つかるとすぐにその個体を出力して終了する．

3.4 テストケースの優先付け

RsRepair において個体の正しさの検証では，生成した個体がテストスイート中の全テストケースを成功するかを確認する．この検証は，個体があるテストケースを失敗することが分かるとすぐにその個体は正しくないと判定する．つまり，テストケースの個体に対する適用回数を削減するためには，テストケースの優先度付けが重要となる．生成した個体が失敗する傾向にあるテストケースの優先順位を上げることで，その他のテストケースの適用を削減可能となる．

RsRepair は Rothermel らが形式化したテストケースの優先度付け問題を以下に示す []．テストスイートを T，PT をテストケースの順列を表す集合，map を PT 内の要素を整数に写像する関数とすると，テストケースの優先度付け問題は以下のように表せる．

$$Problem : \text{Find } T' \in PT \text{ such that } \forall T'' \in PT.\ T'' \neq T' \wedge map(T') \geq map(T'') \tag{2}$$

map は各テストケースの順序付けに評価値を付与する．この関数はテストケースの順序付けの目的によって設定される．例えば，分岐網羅率の観点では，より早く高い分岐網羅率を得るテストの順序付けが高く評価される．RsRepair は map 関数として，より早くソフトウェアの故障を見つけるようなテストケースを高く評価する関数を用いる．ここで，RsRepair 式（2）を実際に解いているのではなく，生成個体に対する過去のテストケース適用結果から近似解を求めていることに注意されたい．

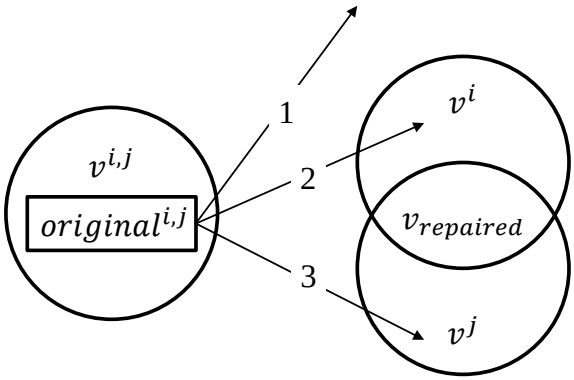


図 2: RsRepair の動作概略

4 研究動機

RsRepair により GenProg の遺伝的アルゴリズムはその計算コストに見合うだけの効果が得られないことが示された．このことは，GenProg の修正能力は遺伝的アルゴリズムによるというよりは，修正対象プログラムに対する変異の操作によることを示す．

しかし，RsRepair には複数の欠陥を含むプログラムを修正できないという問題がある．ここで，複数の欠陥を含むプログラムとは GenProg で定義した変異の操作を複数回行わなければ修正を行えないプログラムを示す．つまり，GenProg や RsRepair において複数の欠陥を含むプログラムとは修正に複数行の変更を要するプログラムを示す．RsRepair は修正対象のプログラムに対するランダムな変異と変異したプログラムの正しさの検証を繰り返し，修正プログラムを探索する．変異したプログラムが正しくない場合は，次の修正候補として修正対象のプログラムを変異したプログラムを生成する．ここで，修正対象のプログラムが修正に複数行の修正を要する場合， RsRepair は修正プログラムを生成できない．図 2 は複数欠陥を含むプログラム $original^{i,j}$ に対する RsRepair の動作を示す．$original^{i,j}$ はプログラムの i, j 行目が欠陥であることを示す．v^i，v^j はそれぞれ修正対象プログラムの j, i 行目を修正した個体を示す．つまり，v^i，v^j はそれぞれ i, j 行目に欠陥を含む．$v_{repaired}$ は修正対象プログラムの 1 i, j 行目の欠陥共に修正した個体を示す．$v_{repaired}$ のみが全テストケースを成功する．また，図 2 のどの円にも囲まれていない空間は円で囲まれた部分の補集合を表す．つまり，i, j 行目以外にも欠陥を含むプログラムや i, j 行目が共に欠陥となるプログラムを表す．このとき，RsRepair は図 2 中の 3 つの矢印で表す動作の内どれかを行う．矢印 1 は i, j 行目いずれの欠陥も修正しない個体を生成する変異を表す．矢印 2，3 はそれぞれ j, i 行目の欠陥を修正する個体を生成する変異を表す．RsRepair が可能ないずれの動作で生成した個体も全テストケースを成功することはない．つまり，RsRepair では複数の欠陥を含むプログラムを修正不可能である．

図 3 に複数の欠陥を含むプログラムの具体例を示す．このプログラムには 2 つの欠陥が存在する．1 つは 3 行目における不必要な代入文で，もう 1 方は 5 行目における return 文の抜けである．この

Algorithm 5 : The multi-RsRepair Algorithm

Input: : Faqulty program P

Input: : Testsuite T including set of positive/negative testcases $PosT$ and $NegT$

Input: : Mutation operator Mutate

Output: : Patch that repairs the faulty program P

1: index $\leftarrow 0$
2: $\{n_0, t_1, t_2, ..., t_n\} \leftarrow T$
3: $T \leftarrow \{(n_0, 1), (t_1, \text{index}), (t_2, \text{index}), ..., (t_n, \text{index})\}$
4: $C_{sub} \leftarrow \text{FaultLocalization}(P, T)$
5: $prev\_variant \leftarrow P$
6: **repeat**
7: $variant \leftarrow \text{Mutate}(prev\_variant, C_{sub})$
8: $Popul \leftarrow \emptyset$
9: **for** $i \leftarrow 0$ to n **do**
10: $(t_{index}, \text{index}) \leftarrow \text{GetTestCase}(T, i)$
11: **if** $\text{PatchValidation}(P, variant, t_{index}) \neq true$ **then**
12: $\text{temp} \leftarrow (t_{index}, \text{index} + 1)$
13: $T \leftarrow \text{Prioritize}(T, \text{temp})$
14: $prev\_variant \leftarrow variant$
15: break
16: **else if** $i = n$ **then**
17: $\text{SuccessFlag} \leftarrow ture$
18: **else**
19: continue
20: **end if**
21: **end for**
22: **until** $\text{SuccessFlag} = true$
23: **return** *variant*

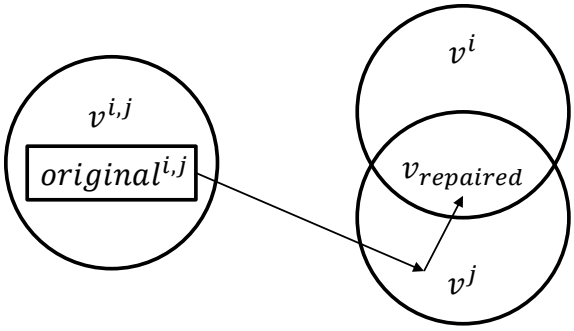
```
 1: void gcd(int a, int b){
 2:   if (a == 0) {
 3:     b = b + 2; // extra statement
 4:     printf("%g¥n", b);
 5:     // return 0; can be repaired by inserting line 18
 6:   }
 7:   while (b != 0) {
 8:     if (a > b) {
 9:       a = a - b;
10:     } else {
11:       b = b - a;
12:     }
13:   }
14:   printf("%g¥n", a);
15:   return 0;
16: }
```

図 3: 複数の欠陥を含む最大公約数を導出するプログラム

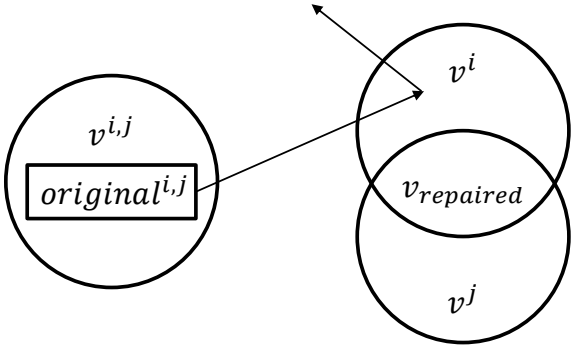
プログラムは delete(3)，add(15, 5) の操作系列で修正可能である．しかし，RsRepair は複数の操作を含む操作系列を生成できない．よって，RsRepair では図 3 を修正不可能である．一方，GenProg は変異の操作系列を重ねて変化させていくので図 3 のようなプログラムも修正可能である．つまり，プログラムの自動修正における遺伝的アルゴリズムに対するランダム探索の優位性は 1 つの欠陥を含むプログラムを修正対象とする場合に限定される．

RsRepair は複数の欠陥を含むプログラムに適用不可能であった．ここで，RsRepair で複数の操作系列を生成することに着目する．RsRepair による複数の操作系列生成のアルゴリズムを Algorithm 5 に示す．このアルゴリズムを multi-RsRepair と呼ぶ．multi-RsRepair は生成した個体が全テストケースを成功しない場合，その個体をさらに変異させて次の個体を生成する．multi-RsRepair は複数の操作系列を含む個体を生成可能なので，複数の欠陥を含むプログラムも修正可能である．しかし，multi-RsRepair はランダムに操作系列を重ねていくので，解から遠ざかるような変異を許容する問題がある．図 4 に multi-RsRepair の動作概略を示す．図 3 と同様に $original^{i,j}$，v^i 及び v^j は i, j 行目に欠陥を含む修正対象プログラム，i 行目に欠陥を含む個体及び j 行目に欠陥を含む個体を表す．さらに，図 4 には i, j, k 行目に欠陥を含む個体 $v^{i,j,k}$ が存在する．よって，各円の補集合である空間は i または j 行目が欠陥でない個体，i, j 行共に欠陥である個体，i, j, k 行目が共に欠陥でない個体を表す．図 4a は複数の欠陥を含むプログラムを適切にする様子を示す．このように，multi-RsRepair は複数の欠陥を含むプログラムを修正可能であるが，ランダム探索が指向性を持たないため問題を生じる．図 4b，4c はランダム探索が指向性を持たないことにより問題が生じる動作を示す．図 4b は一度修正した行を再度欠陥を含む記述にする様子を示す．また，図 4c は修正対象のプログラムに

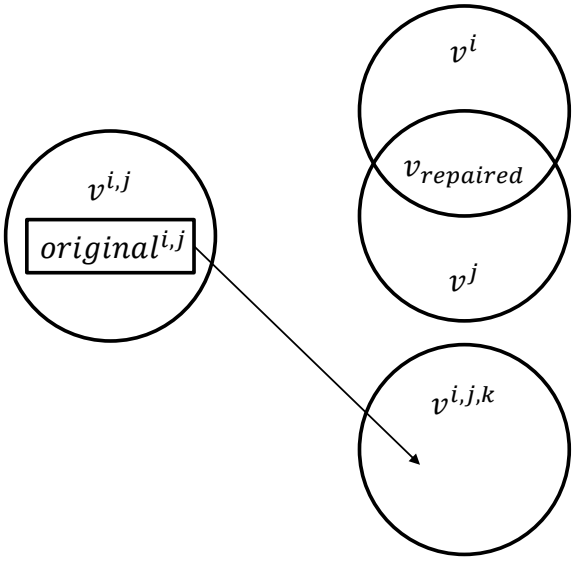
おける欠陥でなかった行を欠陥にする様子を示す．このような問題が存在するため，multi-RsRepair が複数の欠陥を含むプログラムを修正するのは修正した行を再度欠陥を含む記述にすることや，新たに欠陥を含む行を増やすことなしに連続して適切な修正を行うときである．このような，ランダム探索に指向性がないためこのような修正が行われる可能性は低くなる．GenProg では，遺伝的アルゴリズムで変異に指向性を持たせることでこの問題を回避している．しかし，既に述べたように遺伝的アルゴリズムの効果は処理にかかる時間の割りに低いことが指摘されている．そこで，軽量的にランダム探索に指向性を持たせることで，複数の欠陥を含むプログラムを効率的に修正する手法が望まれる．



(a) 複数の欠陥を修正する変異系列



(b) 修正した行を再び欠陥にする変異系列



(c) 新たに欠陥を導入する変異系列

図 4: multi-RsRepair の動作概略

5 提案手法

Algorithm 6 : The Algorithm of Our Proposed Method

Input: : Faulty program P

Input: : Testsuite T including the set of positive/negative testcases $PosT$ and $NegT$

Input: : Mutation operator Mutate

Output: : Patch that repairs the faulty program P

1: $C_{sub} \leftarrow$ FaultLocalization(P, T)
2: $prev\_variant \leftarrow P$
3: **repeat**
4: $variant \leftarrow$ Mutate$(prev_variant, C_{sub})$
5: **for** $i \leftarrow 1$ to $|PosT|$ **do**
6: **if** PatchValidation$(P, variant, PosT_i) \neq true$ **then**
7: break
8: **end if**
9: **end for**
10: $prev\_variant \leftarrow variant$
11: **for** $i \leftarrow 1$ to $|NegT|$ **do**
12: **if** PatchValidation$(P, variant, Neg_i) \neq true$ **then**
13: break
14: **else**
15: append$(PosT, Neg_i)$
16: delete$(NegT, Neg_i)$
17: **end if**
18: **end for**
19: $SuccessFlag \leftarrow true$
20: **until** SuccessFlag $= true$
21: **return** $variant$

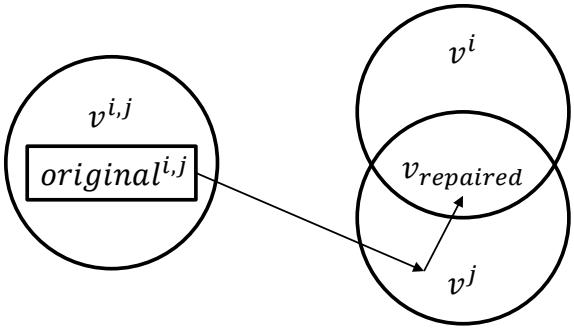
前節で述べたように，ランダム探索により複数の欠陥を含むプログラムを修正することは困難である．そこで，複数の欠陥を含むプログラムを効率的に修正することを目的としてランダム探索に指向性を持たせる手法を提案する．

本研究では，新たに生成した個体が 1 つ前の個体が成功するテストケースを失敗することをそのテストケース破壊するという．提案手法はテストケースを破壊するような変異を無効化することでランダム探索を制御する．テストケースを破壊するような変異を認めないことで，修正したプログラム記述を再び欠陥を含むものに変えることや，新たに欠陥を導入する事を削減する．

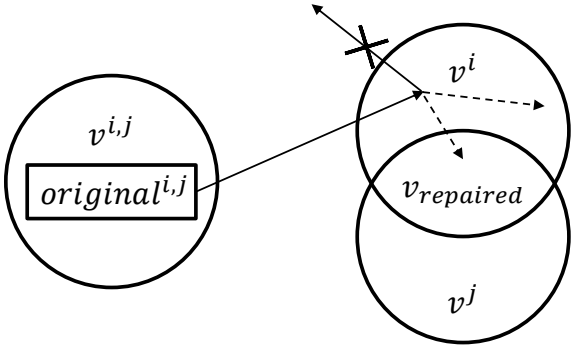
5.1 提案手法のアルゴリズム

Algorithm 6 に提案手法のアルゴリズムを示す．まず，修正対象のプログラムを 1 つ前の個体 $prev_variant$ とする．次に，$prev_variant$ を変異した個体がテストケースを破壊しないかどうかを確認する．3 から 20 行目が修正プログラム探索の繰り返し処理を示す．5 から 9 行目にテストケースの破壊に関する検証を示す．生成した個体がテストケースを破壊しない場合は，$prev_variant$ を更新する．次に，個体の正しさの検証のために失敗テストケースを生成した個体に適用する．全テストケースを成功する場合，生成個体は修正プログラムであるとして処理を終了する．1 つでもテストケースを失敗する場合は，次の個体の生成を行う．

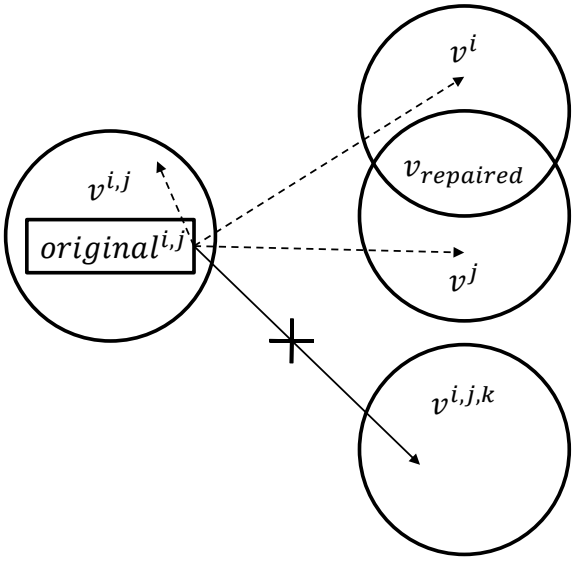
図 5 に提案手法の動作概略を示す．図 5a は複数の欠陥を含むプログラムの修正に関する変異系列を表す．図 5b は修正した行を再び欠陥にすることでテストケースを破壊する変異を無効化する様子を示す．図 5b において破線矢印は許される変異先の空間を表す．図 5c は新たに欠陥を導入することでテストケースを破壊する変異を無効化する様子を示す．図 5c において破線矢印は許される変異先の空間を表す．図 5b，5c より提案手法は変異先を限定していることが分かる．提案手法は変異先の限定により，ランダム探索に指向性を持たせる．



(a) 複数の欠陥を修正する変異系列



(b) 修正した行を再び欠陥にする変異の無効化



(c) 新たに欠陥を導入する変異の無効化

図 5: 提案手法の動作概略

6 実験

本研究では，提案手法の有効性を示すために評価実験を行った．

6.1 実験設定

表 1 に示す実験対象について，提案手法， GenProg，RsRepair 及び Multi-RsRepair を適用，比較することで提案手法の有効性を示す．評価指標として成功率，平均修正時間及び期待修正時間を用いた．成功率はある時間内に修正を完了する可能性を示す．本実験では，15 分を制限時間とし各手法を各実験対象について 50 回ずつ適用した．成功率は 50 回の内 15 分以内に修正を完了した回数の割合を示す．平均修正時間は修正完了時間の平均を示す．つまり，修正を完了しなかった試行は平均修正時間に影響しない．期待修正時間は最初の 1 つの修正を見つけるのにかかるであろう時間を示す．期待修正時間 $expTime$ は成功率 $succRate$ 及び平均修正時間 $aveTime$ を用いて以下の式で表される．

$$expTime = \frac{expTime}{succRate} \tag{3}$$

ソフトウェア開発における欠陥修正では，同一欠陥に対する修正は複数回行われるとは考えにくい．つまり，実際にプログラム自動修正手法がソフトウェア開発に用いられる場合，1 つ修正を発見すれば十分である．そのため，本研究では最初の修正を見つけるまでの時間をプログラム自動修正手法の性能を表す指標とする．

また，本実験では GenProg，Multi-RsRepair 及び提案手法の段階的な変異を 20 に制限した．つまり，20 の変異を重ねても修正が見つからない場合，修正対象のプログラムの変異から再び行う．これは，解が局所最適に陥るのを防ぐためである．

6.2 実験結果

表 2，3 及び 4 にそれぞれ成功率，平均修正時間及び期待修正時間の結果を示す．

表 2 より，ほとんどの場合プロジェクトについて提案手法が 1 番高い成功率となった．また，RsRepair は複数の欠陥を修正できないため gcd 及び tcas-v3 について修正を成功しない．

表 1: 実験対象プロジェクト

プロジェクト	LOC	欠陥の数	欠陥の種類	補足
gcd	25	2	不要なコード，コードの抜け	最大公約数の計算を行う
tcas-v1	173	1	代入文の間違い	飛行機の衝突回避システム
tcas-v2	173	1	条件文の間違い	飛行機の衝突回避システム
tcas-v3	173	2	代入文の間違い，条件文の間違い	飛行機の衝突回避システム

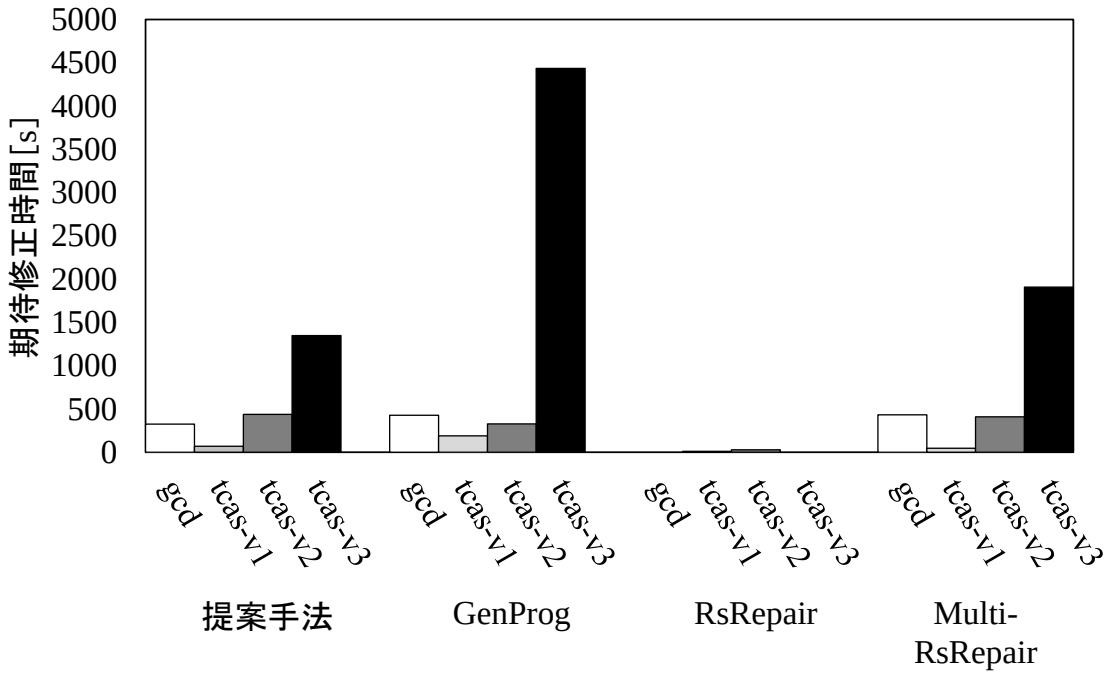


図 6: 期待修正時間の比較

表 3 より欠陥が 1 つの場合は，RsRepair が圧倒的に早く修正を見つけた．提案手法と Multi-RsRepair は tcas-v3 を除いてほとんど変わらない平均修正速度となった．

表 4 より提案手法は複数の欠陥を含むプロジェクトについて他の手法よりも早く最初の修正を見つけると期待できる．図 6 に期待修正時間の比較を示す．提案手法は tcas-v3　及び gcd について他の手法よりも早く最初の修正を見つけることが分かった．

表 2: 成功率の結果

	gcd	tcas-v1	tcas-v2	tcas-v3
提案手法	0.96	1	0.8	0.32
GenProg	0.66	0.78	0.34	0.08
RsRepair	A/N	1	1	A/N
Multi-RsRepair	0.84	1	0.82	0.28

7 考察

表 3 より欠陥が 1 つの場合は，RsRepair が圧倒的に早く修正を見つけたことが分かる．これは，RsRepair が修正完了の判断のみにテストケースを用いるためだと考えられる．Multi-RsRepair も同様に修正の完了の判断のみにテストケースを用いるが完全にランダムな変異を段階的に行うため，解から遠ざかっていくことが原因だと考えられる．

表 4， 図 6 より提案手法は複数の欠陥を含むプロジェクトについて他の手法よりも早く最初の修正を見つけることがわかる．このことから，テストケースを用いた軽量的な指向性制御に効果があったと考えられる．修正する欠陥が 1 箇所の場合は，指向性の制御の必要がないため，他の手法よりも若干遅くなる．しかし，GenProg と提案手法では，提案手法のほうがほとんどのプロジェクトについて早く修正可能なことから， 遺伝的アルゴリズムよりも効果的に修正を制御していることがわかる．

提案手法と Multi-RsRepair は tcas-v3 を除いてほとんど変わらない平均修正速度となった．これは，対象プロジェクトのサイズが原因だと考えられる．対象プロジェクトが大きくなると，修正の探索空間も大きくなるため，制御なしに変異を重ねると見当違いの修正を行う可能性が高くなる．このことは，複数のバグを含むプロジェクト gcd と tcas-v3 の結果を比較することで確認できる．

また，提案手法は成功テストの破壊により変異を無効化しランダム探索を制御するので，提案手法はテストケースの質に依存すると考えられる．網羅率が高いテストスイートを用いると間違った変更が及ぼす影響を補足する可能性が高くなる．つまり，間違った変異を無効化する可能性が高くなる．

表 3: 平均修正時間の結果

	gcd	tcas-v1	tcas-v2	tcas-v3
提案手法	311.96	70.52	350.51	431.28
GenProg	283.00	148.46	112.05	354.73
RsRepair	A/N	10.40	30.15	A/N
Multi-RsRepair	363.29	47.28	336.78	534.66

8 関連研究

GenProg や RsRepair は ” 既にプログラム中に存在する記述を用いることでプログラム中の欠陥を修正可能である “という仮定のもとに動作する．つまり，この仮定が正しくなければ GenProg や RsRepair はプログラムの修正を適切に行えない．そこで，Barr らはバージョン管理システムで開発されたプロジェクトにおいて仮定の正しさを検証した [26]．検証の結果，開発者がコミット間に追加したソースコードの内，約 40% ものプログラム記述が既存のプログラム中に存在する記述を用いることで開発可能であった．また， Martinez らも同様にバージョン管理システムで開発されたプロジェクトにおいて仮定の正しさを検証した [27]．Barr らはプログラム記述のまとまりを行単位と限定したが，Martinez らは行単位のまとまりに加えて，トークンに関する記述のまとまりに関して調査した．

プログラムの自動修正に関して様々な研究がなされている．Kim らはオープンソースソフトウェアに関する調査からプログラムの修正パターンを 10 種類に分類し，分類パターンに基づき修正候補を選択することで修正候補を限定した [28]．また，Ji n らは並列プログラミングに対応したプログラム自動修正手法 A-Fix を提案した [29]．並列システムでは各プロセスが非決定的な動作をするため、特殊な欠陥同定技術が必要となる．そこで， A-fix は欠陥同定を CTrigger [30] を用いて行った．

プログラム自動修正に関して，様々な研究が存在する．プログラム自動修正の分野において評価指標が明確に定まっていない．そこで，各研究者は独自の評価指標を用いて提案手法を評価することが多い．Monperrus らはプログラム自動修正において少なくとも述べなくてはならない点として，修正可能な欠陥種類を挙げるべきと述べた [31]．本研究は，既存のプログラム記述に基づきプログラムの自動修正を行う．よって，開発済みのプログラム中に存在しないプログラム記述を用いた修正を行うことができない．null チェックやバッファオーバーフローを防ぐためのプログラム記述はある程度決まっている．つまり，本研究で提案した手法はプログラムのコントロールフローを換えるような修正や，複雑な計算の導入などは行えないが，null チェックやバッファオーバーフローの修正は可能である．

表 4: 期待修正時間の結果

	gcd	tcas-v1	tcas-v2	tcas-v3
提案手法	324.96	70.52	438.14	1347.71
GenProg	428.79	190.33	329.58	4434.15
RsRepair	A/N	10.40	30.15	A/N
Multi-RsRepair	432.48	47.28	410.70	1909.51

9　まとめと今後の課題

本論文では，テストケースで指向性をもたせたランダム探索によるプログラム自動修正手法を提案した．既存手法である GenProg は計算コストが大きいという欠点が，RsRepair には複数の欠陥を含むプログラムを修正できないという欠点があった．RsRepair は修正を段階的に行わないため，複数の欠陥を含むプログラムを修正不可能である．しかし，RsRepair は完全にランダムな探索でプログラムを変異するので，制御無しに段階的変異を行うと見当違いのプログラムを生成するという問題点があった．

そこで，本研究ではランダム探索によるプログラム変異の妥当性の検証をテストケースを用いて行った．この検証は最悪でも GenProg と同等の計算コストである．評価実験により，提案手法は複数の欠陥を含むプログラムを GenProg よりも短い時間で修正可能なことを示した．

プログラム自動修正において，修正候補の限定は重要な課題である．本研究では，欠陥と疑わしいプログラム記述に関する限定は行っているが修正記述候補に関する限定は行っていない．そこで，より効率的なプログラムの自動修正に向けて修正記述の候補を限定する手法の考案を今後の課題とする．また，本手法では用いた修正記述を既に開発済みのプログラム記述としているため，プログラム記述に存在しない式による修正が不可能である．そこで，新たなプログラム記述を生成する手法を今後の課題とする．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究の全過程を通して， 熱心かつ丁寧なご指導を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して，実験環境の整備に関する的確な助言を頂きました 井垣 宏 特任准教授 に深く感謝申し上げます.

本研究を行うにあたり，日常の議論の中でご助言を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等で多くの知識や示唆を頂きました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] J Baker. Experts battle £192bn loss to computer bugs. http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html, Feb 2012. Accessed 2015-01-27.

[2] T Britton, L Jeng, GT Carver, P Cheak, and T Katzenellenbogen. Reversible debugging software. Technical report, Univercity of Cambridge, Judge Buisiness School, 2013.

[3] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pp. 215–222, Sept 1976.

[4] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference onSoftware Engineering, 2007.* (ICSE 2007), pp. 75–84, May 2007.

[5] Carlos Pacheco. *Directed Random Testing.* Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 2009.

[6] R. Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering, 2007.* (ICSE 2007), pp. 416–426, May 2007.

[7] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24rd International Conference on Software Engineering, 2002.* (ICSE 2002)., pp. 467–477, May 2002.

[8] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 45–55, Washington, DC, USA, 2009. IEEE Computer Society.

[9] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 89–98, Sept 2007.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[13] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 254–265, New York, NY, USA, 2014. ACM.

[14] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[15] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[16] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pp. 1–10, New York, NY, USA, 2002. ACM.

[17] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 467–477, New York, NY, USA, 2002. ACM.

[18] A Eiben and J Smith. *Introduction to Evolutionary computing*. Springer, 2003.

[19] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pp. 965–972, New York, NY, USA, 2010. ACM.

[20] L Miller.B and D Goldberg E. Genetic algorithms, selection schemes, and the varying effects of noise. *Evol Comput*, Vol. 4, No. 2, pp. 113–131, 1996.

[21] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 1–10, May 2011.

[22] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, Vol. 45, No. 1, pp. 11:1–11:61, December 2012.

[23] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Empirical software engineering and verification. chapter Search Based Software Engineering: Techniques, Taxonomy, Tutorial, pp. 1–59. Springer-Verlag, Berlin, Heidelberg, 2012.

[24] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 180–189, Sept 2013.

[25] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, Vol. 22, No. 2, pp. 67–120, March 2012.

[26] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 306–317, New York, NY, USA, 2014. ACM.

[27] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pp. 492–495, New York, NY, USA, 2014. ACM.

[28] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[29] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 389–400, New York, NY, USA, 2011. ACM.

[30] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pp. 25–36, New York, NY, USA, 2009. ACM.

[31] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software

repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 234–242, New York, NY, USA, 2014. ACM.