修士学位論文

題目

遺伝的プログラミングにおける選択的交叉を用いた
自動プログラム修正手法

指導教員
楠本 真二 教授

報告者
高　良多朗

平成 28 年 2 月 9 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 27 年度 修士学位論文

遺伝的プログラミングにおける選択的交叉を用いた
自動プログラム修正手法

高　良多朗

内容梗概

ソフトウェアの開発工程および保守工程においてはそのソフトウェアの信頼性や安全性を保証する必要があるため，デバッグは非常に重要な作業である．しかし，ソフトウェアは開発を通して大規模化および複雑化する傾向にあり，同様にデバッグに要する人的資源や時間的資源が増大して開発工程全体に悪影響を及ぼす場合がある．このような問題を解決するためにデバッグの効率化を目的とした手法，特に自動プログラム修正手法が注目を浴びている．近年では遺伝的プログラミングを用いて修正プログラムを生成する自動プログラム修正手法がその性能から高い評価を得ており，様々な発展手法がまた提案されている．この手法は大きく分けて選択・変異・交叉の 3 つの処理によって修正プログラム候補を生成する．既存の発展手法は主に選択処理や変異処理について着目して発展させているが，本研究では一度の処理で大きく変化した修正プログラム候補を生成できる交叉処理に着目した．既存手法のように交叉対象となる修正プログラム候補を無作為に決定するのでなく，定量的な基準を用いて交叉対象を決定して交叉処理を実行することでより優れた修正プログラム候補を生成できると考えた．本研究では，修正プログラム候補間での特徴量の比較によって交叉対象を選択する，選択的交叉手法について提案する．既存手法との比較実験を行った結果，提案手法は既存手法に比べて修正の成功率が高いことが示された．

主な用語

ソフトウェア保守
ソースコード解析
遺伝的プログラミング
自動プログラム修正

目 次

1 まえがき **1**

2 準備 **3**

2.1 テストケースを用いたデバッグ . . . 3

2.2 欠陥箇所の限局 . . . 4

2.3 遺伝的プログラミング . . . 4

2.4 GenProg . . . 5

2.4.1 選択処理 . . . 6

2.4.2 変異処理 . . . 6

2.4.3 交叉処理 . . . 6

3 関連研究 **8**

3.1 テストケースの自動生成 . . . 8

3.2 欠陥箇所の限局 . . . 8

3.3 自動プログラム修正 . . . 8

3.4 交叉処理の特色 . . . 9

4 提案手法 **11**

4.1 個体間評価値の計算 . . . 12

4.1.1 類似度での評価 . . . 12

4.1.2 テストケースでの評価 . . . 13

4.1.3 双方での評価 . . . 13

4.2 交叉対象の選択 . . . 13

4.2.1 評価値を用いた選択 . . . 13

4.2.2 秀逸個体を中心にした選択 . . . 14

4.3 交叉の実行 . . . 15

4.3.1 一点交叉 . . . 15

4.3.2 一様交叉 . . . 16

4.3.3 ランダム交叉 . . . 16

4.3.4 適合値交叉 . . . 16

5 評価実験 **17**

5.1 実験対象 . . . 17

5.2 実験方法 . . . 17

5.3 実験設定 18
5.4 実験結果 19

6 考察 **24**
6.1 個体間評価方法 24
6.1.1 類似度での評価 24
6.1.2 テストケースでの評価 26
6.1.3 双方での評価 26
6.2 交叉方式 27
6.2.1 一点交叉 27
6.2.2 一様交叉 27
6.2.3 ランダム交叉 27
6.2.4 適合値交叉 28
6.3 秀逸個体の有無 28

7 妥当性の脅威 **29**

8 あとがき **30**

謝辞 **31**

参考文献 **32**

図 目 次

1 テストケースを用いたデバッグの例 3
2 GenProg の流れ 5
3 提案手法における交叉処理の流れ 11
4 個体間評価方法 12
5 秀逸個体の有無による交叉処理の変化 14
6 交叉方法 15
7 実行時間 22
8 修正世代数 23
9 バージョン 9 における実行時間と修正世代数の分布図 24
10 バージョン 43 における実行時間と修正世代数の分布図 25

1　まえがき

デバッグ，すなわちソフトウェア中に生じた欠陥の原因 (バグ) を特定して修正する工程はソフトウェアの開発工程および保守工程において必要不可欠である．なぜならば，ソフトウェア品質において信頼性および安全性，すなわち欠陥の発生頻度や欠陥による影響の大きさは非常に重要な項目であり，これらが欠如したソフトウェアを利用した場合は多大な損害を招く可能性がある．過去には，病院で用いられていた管理システムが計算ミスを引き起こし，これが原因で複数名の命が失われた事例が報告されている [1].

以上の背景から，欠陥を可能な限り修正するために熟練の開発者がデバッグを行うことが大半だが，開発を通して大規模化および複雑化したソフトウェアに対するデバッグは非常に長い時間を必要とする．Baker の調査では開発工程の半数以上がデバッグに費やされた事例が確認されている [2]．それ以外では，日に 300 以上もの欠陥が生じて手動では対処しきれなかった事例や [3]，単一のバグを修復するために半年以上を要した事例 [4]，中には公開して数年が経過してから欠陥の存在が発覚した事例も存在する [5].

上述の背景から，デバッグの支援を目的とした手法が多々提案されている．欠陥の発見およびバグの特定を容易にするためにテストケースを自動生成する手法や [6, 7]，ソースコードの内容やテストケース実行時の挙動をメトリクス化して欠陥箇所を限局する手法など [8, 9, 10]，様々な観点から手法が提案されている．近年ではデバッグの補助のみにとどまらず，バグの特定から修正に加えて成否の確認までを自動で行う自動プログラム修正手法が注目を集めており，中でも GenProg[11] を始めとした，遺伝的プログラミングに基づいて修正プログラムを生成する手法が高く評価されている．

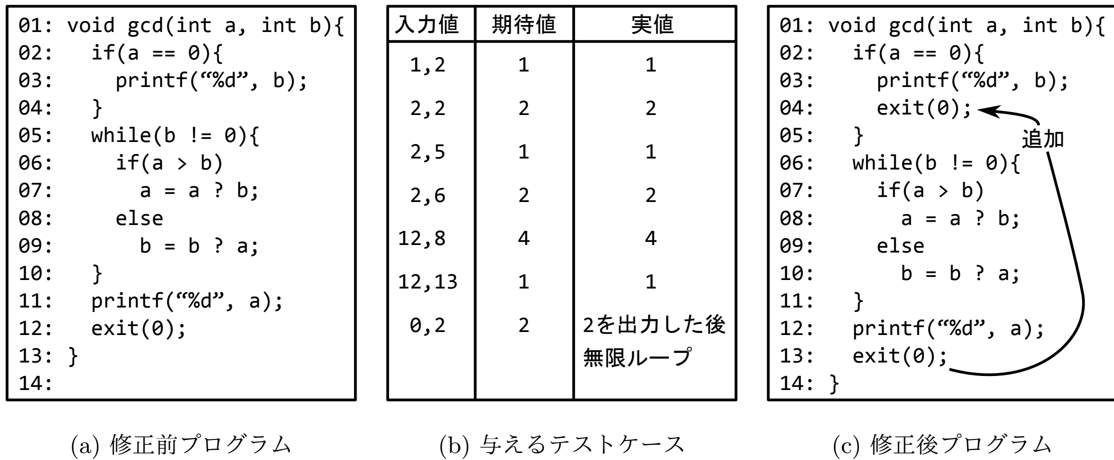
遺伝的プログラミングは生物の進化過程を模した探索アルゴリズムであり，個体の適合値を評価して優れたものを一定数取り出す “選択”，プログラムを変形して個体を生成する “変異”，複数の個体を組合わせて新しい個体を生成する “交叉” で構成される．GenProg およびその関連手法では修正プログラム候補を生物の個体と見立てて，テストケースの実行結果を用いて修正プログラム候補の評価や変形を行う．

GenProg の関連手法としては，ランダム探索によって修正プログラムを生成する RSRepair[12] や, あらかじめ用意した修正パターンを用いてプログラムを変形する PAR[13]，与えられた制約式を満たすようにプログラムを変形する SemFix[14] および DirectFix[15] などが存在するが，いずれも選択処理または変異処理に着目した手法である．

一方で本研究では交叉処理の特色である，一度の処理で大きな変化を引起こせる能力に着目した．本研究では，交叉対象となる修正プログラム候補を類似度や適合値といった基準を用いて選択する手法に加えて，新たな交叉方式すなわち交叉で入換える要素を選択する手法を組み合わせた選択的交叉を提案する．また，手法の有効性を評価するために GenProg の

交叉処理に上述のアルゴリズムを実装したツール “Marriagent” を作成して GenProg との比較実験を行った．実験結果より，Marriagent は GenProg に比べて修正の成功率が高いことを示した．

以降，2 章でデバッグの支援手法，特に GenProg について紹介した上で 3 章で関連研究及び本研究の動機について述べ，4 章で提案手法について説明する．5 章では提案手法の評価実験について説明して 6 章で実験結果の考察を行う．7 章で妥当性の脅威について述べた後，最後に 8 章でで本研究のまとめと今後の課題について述べる．



```
01: void gcd(int a, int b){
02:   if(a == 0){
03:     printf(“%d”, b);
04:   }
05:   while(b != 0){
06:     if(a > b)
07:       a = a ? b;
08:     else
09:       b = b ? a;
10:   }
11:   printf(“%d”, a);
12:   exit(0);
13: }
14:
```

(a) 修正前プログラム

入力値	期待値	実値
1,2	1	1
2,2	2	2
2,5	1	1
2,6	2	2
12,8	4	4
12,13	1	1
0,2	2	2を出力した後無限ループ

(b) 与えるテストケース

```
01: void gcd(int a, int b){
02:   if(a == 0){
03:     printf(“%d”, b);
04:     exit(0);   ← 追加
05:   }
06:   while(b != 0){
07:     if(a > b)
08:       a = a ? b;
09:     else
10:       b = b ? a;
11:   }
12:   printf(“%d”, a);
13:   exit(0);
14: }
```

(c) 修正後プログラム

図 1: テストケースを用いたデバッグの例

2 準備

2.1 テストケースを用いたデバッグ

一般的に，デバッグとはソフトウェア中に生じた欠陥の発見から始まり，その原因すなわちバグを特定して修正し，最後に修正の成否を確認するまでを指す．デバッグを行うためにはまずバグが存在する箇所，言い換えればソースコードの実装や機能的要求などの誤りを特定する必要がある．

最も簡単にバグを特定できる手段にテストケースの使用が挙げられる．テストケースとは入力値，期待値，実値の組を指し，ある入力値に対する出力値 (実値) が要求される値 (期待値) と等しいか否かを判定するものである．複数のテストケースを実行した上で，どの入力値を与えた場合に誤った動作となるかを把握することでバグを特定し易く，言い換えればすべてのテストケースが正しく動作するならばそのソフトウェアは信頼性が高いと判断できる．理想的なテストケースはソフトウェアの全動作パターンを確認するものであるが，そのようなテストケースの作成や実行は膨大な時間や労力を必要とするため現実的ではない．したがって，実践上では到達可能な実行パスを網羅する程度に留める場合が大半である．しかし，その場合でも十分なテストケースを手動で用意するには多大な労力を要するため，テストケースを自動で生成する手法が提案されている．

以降，本研究ではデバッグ開始時のソフトウェアが通過するテストケースを**通過済テスト**と呼び，通過しないテストケースを**未通過テスト**と呼ぶ．言い換えれば，通過済テストは正しい挙動を示し，未通過テストは誤った挙動すなわち欠陥を示す．

2.2　欠陥箇所の限局

デバッグを支援する主な手法に，**欠陥箇所の限局** (Fault Localization) が挙げられる．欠陥箇所の限局手法はソースコード解析を通してバグの候補を探す手法であり，最もバグの可能性が高い箇所を開発者に提示することでデバッグを補助する．中でもテストケース毎の実行パスから算出した疑惑点に基づいて順序付けを行う手法が存在する [16]．直感的に，通過済テストのみが実行する文はバグを含む可能性が低く，未通過テストのみが実行する文はバグを含む可能性が高いと推測できる．図 1 にテストケースを用いて欠陥箇所の限局を行う例を示す．図 1(b) は図 1(a) に与えるテストケースの集合で，最下部の入力 (0,2) が未通過テストで，それ以外が通過済テストとなる．この例では，図 1(a) の中で未通過テストのみが通過する行は 3 行目であるため，修復箇所は 3 行目の近傍だと推測される．図 1(c) は修正の一例で，3 行目の直後に文を追加して無限ループを防いでいる．また，GenProg およびその関連手法のうちいくつかはプログラムの変形を行う際，同様にテストケースの実行パスを調べて修正箇所を決定する．

2.3　遺伝的プログラミング

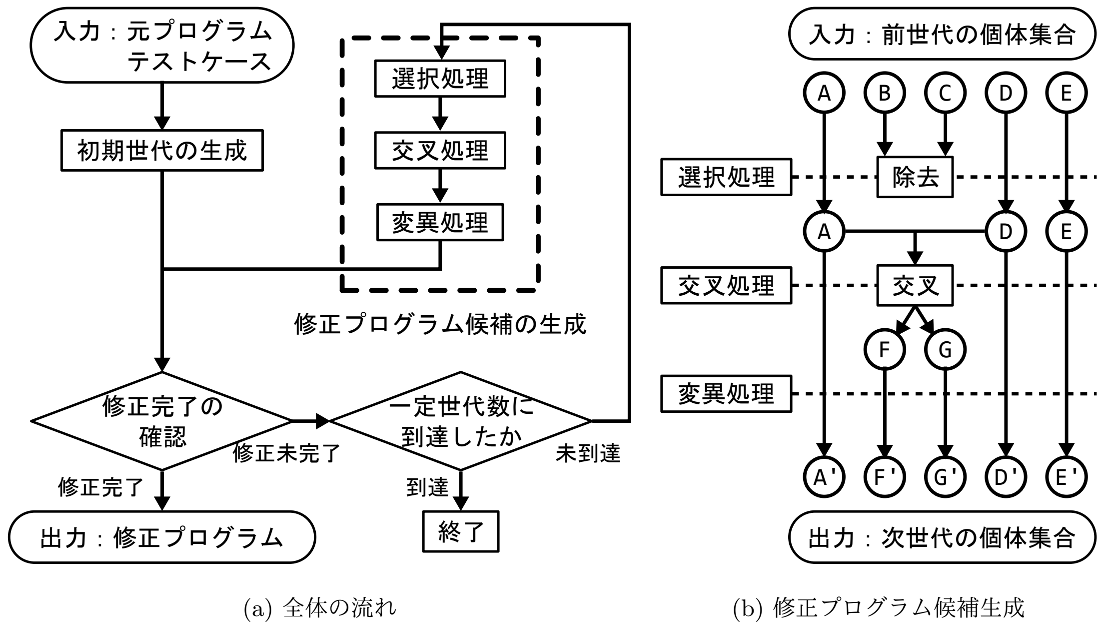
遺伝的プログラミング (Genetic Programming) とは生物の進化過程を模した探索アルゴリズムで，プログラムを生物の個体に見立てたものである [17]．このアルゴリズムは個体の集合を 1 世代として個体の変形や生存選択を周期的に繰り返し，徐々に解に収束させることを目的としている．遺伝的プログラミングの主要部分は以下 3 つの処理から成り立つ．

選択　各個体の適合値を評価して一定数の個体を取り出す．適合値は環境，すなわち解への近さを示す評価値で，これが高いほど優れた個体とみなす．

変異　各個体ごとに変形を加え，次世代の個体を生成する．この操作で加える変形は操作対象や手法によって大きく異なり，ビット単位での変更やソースコードの書き換えなど多岐にわたる．

交叉　2 つの個体を選び，各個体の要素を半分ずつ併せ持つ個体を生成する．変異処理と同様に，個体要素は操作対象や手法によって大きく異なり，ビット列を入換えるものもあれば構文木の一部を入換えるものもある．

遺伝的プログラミングの終了条件はある個体の適合値が最大に到達した場合で，すなわち完全な解が得られた場合はその個体を出力する．または，一定数の世代に達した場合や一定時間が経過した場合は完全な解が得られないと判断して処理を終えることが多い．



(a) 全体の流れ

(b) 修正プログラム候補生成

図 2: GenProg の流れ

2.4 GenProg

遺伝的プログラミングを用いた自動プログラム修正手法に GenProg[11] が存在する．GenProg は修正対象のプログラムとテストケースを入力とし，図 2 に示す流れで修正プログラム候補を生成する．生成した修正プログラム候補のいずれかが全テストケースを通過する場合，そのプログラムを修正プログラムとして出力する．修正プログラム候補の生成は図 2(b) に示す流れで生成される．各処理の概要を以下に示す．

選択 生成された各修正プログラム候補に対してテストケースを実行し，多くのテストケースを通過するプログラムから順に一定数を取り出す．

変異 各修正プログラム候補にさらなる変形を加え，次世代の修正プログラム候補を生成する．プログラムの変形は追加，削除，置換からなる．

交叉 2 つの修正プログラム候補を選び，各々に加えられた変形を半分ずつ入換える．生成に使用した修正プログラム候補は取り除かず，そのまま次世代の生成に使用する．

ここで，GenProg における個体の構成要素はプログラムの文や字句ではなく，“x 行目を y 行目に追加” の様に，変異処理によって加えられた変形内容であることに留意されたい．つまり，元のプログラムは要素を持たない個体であり，交叉処理で交換する対象は変形内容の集合である．これらの変形内容は順不同であり，順番を入換えても実行結果には影響しない．

以下，各処理についてその詳細を述べる．

2.4.1　選択処理

選択処理では，各修正プログラム候補の適合値を計算する．適合値は式 (1) で計算される．

$$\text{適合値} = W_{PosT} \times |P_{PosT}| + W_{NegT} \times |P_{NegT}| \tag{1}$$

$$\left.\begin{aligned} W_{PosT} &= 1 \\ W_{NegT} &= 2 \times \frac{|PosT|}{|NegT|} \end{aligned}\right\} \tag{2}$$

$$\text{最大適合値} = |PosT| \times 3 \tag{3}$$

- $PosT$：通過済テストの集合
- $NegT$：未通過テストの集合
- W_x：x の重み
- P_x：通過可能な x

式 (1) で重みを設定する理由は二種類のテストの重要性にあり，通過済テストを通るプログラムよりも，未通過テストを通るプログラムの方が解に近いと判断されたためである．GenProg のデフォルト設定では各テストケースの重みは式 (2) で与えられる．式 (2) に従った場合，式 (1) で得られる最大適合値は式 (3) となる．

次に，適合値が高い修正プログラム候補から順番に一定数を取り出す．取り出す数は実行時設定で与えられ，世代によって変化しない．

2.4.2　変異処理

変異処理では，各修正プログラム候補に対して新たな変形を加える．変形は追加，削除，置換の 3 つの操作からなり，未通過テストの実行パス上のどこかに対して行われる．

また，追加，置換においては，実行パス上の文をランダムに抽出して再利用している．欠陥箇所の限局手法の例で示した図 1 では終了文を 3 行目の直後に追加することでバグを修正しているが，この文は 12 行目に存在するため，GenProg は同様の修正が可能である．らの調査ではソースコードに加えられた修正の内，約半数が再利用可能であり，この方法は有効であると報告されている [18].

2.4.3　交叉処理

交叉処理では，2 つの修正プログラム候補を無作為に選び，各プログラムに与えられた変形を半分ずつ入換える．この時，新たな変形を加えることはない．

GenProg では入換え方法に一点交叉を使用している．この方法は要素配列を中間で区切り，後半同士を入換える方法である．入換え方法については提案手法の章，節 4.3 で具体例を示す．

3　関連研究

3.1　テストケースの自動生成

テストケースの自動生成手法は EvoSuite[6] が有名である．この手法は様々な入力値をランダムに生成して与えていき，可能な限り少数かつ高網羅率のテストケース集合を生成する手法である．EvoSuite が与える入力値は乱数によって生成されるランダム値の他に，バイトコード中に表れるリテラルを再利用する場合がある．これは特定の値，特に文字列を与えた場合のみ起こる分岐を網羅するためである．また，ランダム生成以外に遺伝的プログラミングも使用可能であり，設定次第で様々な生成パターンを使用可能である．

3.2　欠陥箇所の限局

欠陥箇所の限局手法の代表としては Tarantula[8] が存在する．この手法はソースコード中の実行文に対して式 (4) で示される疑惑値を計算して順序付けを行い，疑惑値が高い実行文から順番に開発者に提示する．この手法に限らず，テストケースを用いる欠陥箇所の限局は使用するテストケースに強く依存するため，同一ソフトウェアを対象にしても異なるテストケースを用いれば実行結果が変化する場合がある．

$$\text{疑惑値} = \frac{\text{未通過テストの実行率}}{\text{通過済テストの実行率} + \text{未通過テストの実行率}} \tag{4}$$

$$\text{テストケースの実行率} = \frac{\text{その実行文を通るテストケースの数}}{\text{テストケースの総数}} \tag{5}$$

MULTRIC[9] は 25 種類の順序メトリクスを組合わせる欠陥箇所の限局手法であり，どの順序メトリクスをどの重み付けで用いるかを機械学習を用いて決定する．この手法の有効性は学習モデルに強く依存するが，使用対象ソフトウェアの立上げ時からの実行履歴を学習モデルに用いることで，そのソフトウェアに適した順序メトリクスを用いて欠陥箇所を特定することが可能である．

MIMIC[10] は通過済テストによる正常な挙動をモデル化することで未通過テストの異常な挙動を監視する．この手法の特徴として，未通過テストが違反するモデルを開発者に提示することでどのような修正を行うべきかの指標を示すことが出来る点が挙げられる．例を挙げると，“$a > 0$” や “$x == null$” といったモデルは欠陥の特徴をそのまま示すことが多いため，開発者は容易に修正法を特定できる．

3.3　自動プログラム修正

自動プログラム修正手法は GenProg のアルゴリズムを発展させたものがいくつか存在している．RSRepair[12] は遺伝的プログラミングの代わりにランダム探索を採用しており，世

代更新を行わずに一箇所のみの変異を行った修正プログラム候補の生成を繰り返す．この手法の特徴として，修正成否の確認中にいずれかのテストを通過しなかった場合はその時点で確認を中止して新たな修正プログラム候補を生成することで不要なテスト実行を大きく削減できる．このため，RSRepair は一箇所の変異で修正可能なバグに対しては非常に優れた性能を持つが，複数個所の修正が必要なバグは修正不可能という欠点を持つ．

PAR[13] はあらかじめ修正パターンを用意しておき，それらを用いてプログラムの変形を行う．定型的な修正を用いることで多くのソフトウェアに生じるバグの修正が容易であり，また修正プログラムの理解性の高さも評価されている．一方で，修正対象ソフトウェア特有の記述を要するバグに対応できない欠点を持つため，後述の Prophet のように修正パターンをツール側で作成する発展手法が提案されている．また，手法の有効性及び実験の妥当性について別著者による批判的議論が行われている [19].

SemFix[14] はテストケースから満たすべき制約式を生成し，これらを全て満たすようにプログラムを変形する．SemFix の最大の特徴として，既存のプログラム文を再利用するという方針ではないため，過去に記述されていない文を使用できる点が挙げられるが，制約式の解は NP 困難であり，場合によっては実現不可能な制約式が生成されたり，計算時間が膨大になることもある．SemFix を発展させた手法には DirectFix[15] が存在し，より簡潔で理解し易い修正が可能な点が評価されているが，SemFix と同様に制約式の計算時間については課題を残している．

SPR[20] は条件式や値域を徐々に狭めていくことで，膨大な探索範囲を効率よく調査する Staged Program Repair と名づけられたアルゴリズムを使用している．探索空間の限局により，SPR は既存手法よりも正確な修正が行いやすいと報告されている．

Prophet[21] は PAR のように人の手による修正パターンを利用する．この手法は与えられた修正パターンをモデル化し，このモデルに適合するようにプログラムの変形を行うため，高い修正のみならず，修正の妥当性も保障されている．

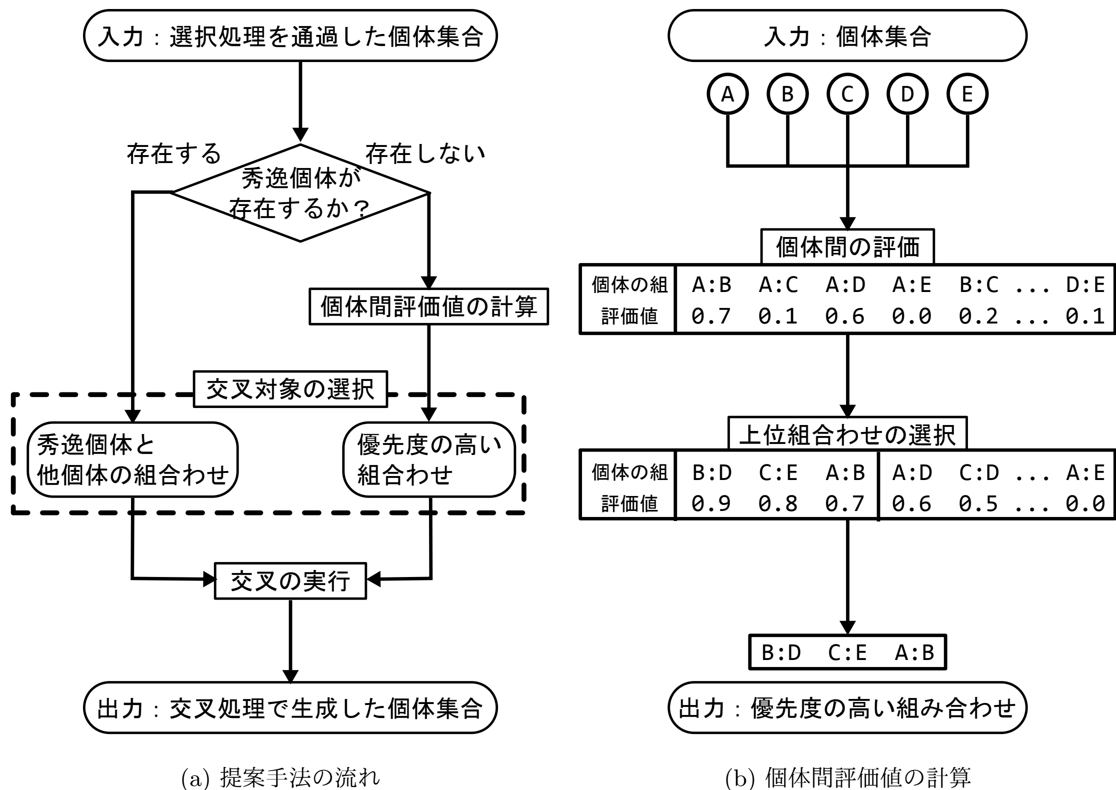
Le らの手法 [22] は過去のプロジェクトの修正履歴を調査し，実際に行われた修正パターンを再利用することで PAR や Prophet 同様，実践上で行われる記述に近い修正が可能である．

3.4　交叉処理の特色

節 2.4 で挙げた自動プログラム修正手法はいずれも修正プログラム候補の評価またはプログラムの変形，言い換えれば選択処理や変異処理を改良した手法である．一方で，複数のプログラムを組合わせる交叉処理も他処理と同様に十分な改良の余地を残している．そこで，本研究では交叉処理の持つ特色に着目した．プログラムの変形を行う交叉処理と変異処理を比較すると，交叉処理は以下に示す特色を持つ．

- 一度の処理で複数の要素を入換えるため，大きな変化を起こしやすい
- 過去に加えられた不要な変形を取消すことが可能
- 同じプログラムの組を用いた場合でも，交叉パターンを変えれば異なるプログラムを生成できる

上述の特色を活かすように修正プログラム候補の組合わせを選んで要素を入換えることで，必要な変形のみを加えた修正プログラムが生成できると考えた．以上から，本研究では交叉対象の選択方法および交叉方式を改良した手法を提案する．



(a) 提案手法の流れ

(b) 個体間評価値の計算

図 3: 提案手法における交叉処理の流れ

4　提案手法

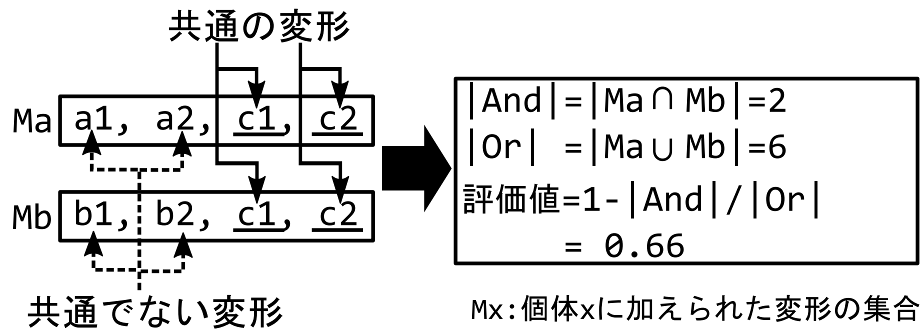
提案手法の流れを図 3 に示す．提案手法全体では GenProg と同様に図 2 に示す流れで修正プログラム候補の生成を行うが，交叉処理は図 3(a) に示す流れとなる．選択処理および変異処理は GenProg と同一のアルゴリズムを使用しており，これらは節 2.4.1 および節 2.4.2 で説明している．提案手法は主として以下に示す 3 つのステップから成る．

個体間評価値の計算 最新世代の個体の全組合せについて，指定の評価方法を用いて交叉処理の優先度を計算する．本ステップは図 3(b) に示す流れで行う．

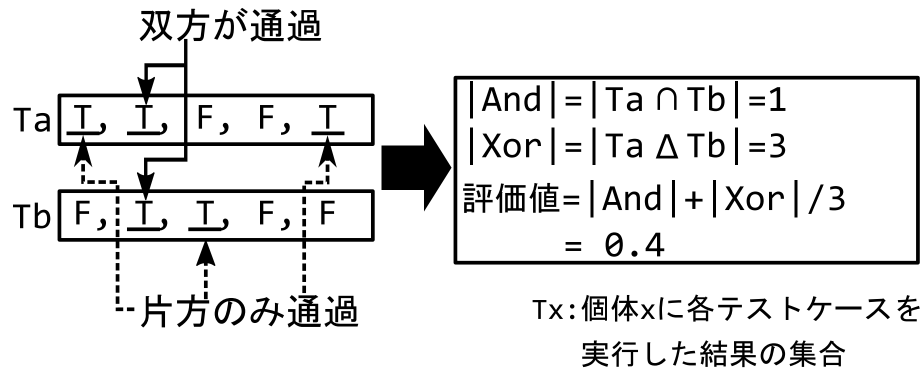
交叉対象の選択 交叉処理の優先度を用いて交叉対象の組合わせを選択する．特に適合値の高い秀逸個体が存在する場合，その個体と他全個体の組合わせ交叉対象に選択する．

交叉の実行 各交叉対象について指定の方法で交叉処理を実行し，新たな個体を生成する．既存手法と同様に生成に使用した修正プログラム候補は取り除かず，そのまま次世代の生成に使用する．

次に，各ステップについてその内容を詳細に述べる．



(a) 類似度での評価



(b) テストケースでの評価

図 4: 個体間評価方法

4.1 個体間評価値の計算

提案手法では図 4 に示すように 2 種類の個体間評価方法を使用する．

4.1.1 類似度での評価

一度の処理で大きな変化が狙えるという交叉の特色を活かすためには，できるだけ異なる要素を持つ個体を選択することが望ましい．したがって，図 4(a) に示す方法では個体間での類似度を評価する．類似度の計算方法は式 (6) に示す Jaccard 係数を用いる．提案手法では類似度が低いほど良い組合わせとみなすため，式 (7) の値で評価する．この評価値は [0..1] の範囲をとる．

$$Jaccard\,係数 = \frac{|M_a \cap M_b|}{|M_a \cup M_b|} \tag{6}$$

$$類似度評価値 = 1 - Jaccard\,係数 = 1 - \frac{|M_a \cap M_b|}{|M_a \cup M_b|} \tag{7}$$

- M_a, M_b: プログラム a, b に加えられた変形の集合
- $A \cap B$: A と B に共通して加えられた変形の集合

- $A \cup B$: A と B の片方のみに加えられた変形の集合
- $|A|$: A の要素数

4.1.2 テストケースでの評価

交叉処理によって優れ個体を生成するためにはより優れた個体同士を選ぶことも重要である．そこで，図 4(b) に示す方法では各個体に対するテストケースの実行結果を比較し，より多くのテストケースを通過する個体同士の組合わせほど良いとする．この評価方法ではテストケース毎に双方通過 (And)，片方通過 (Xor)，双方失敗のいずれか判定を行い，式 (8) の値を評価する．この式の値は類似度と同様に [0..1] を取る．

$$\text{テストケース評価値} = \frac{|T_a \cap T_b| + |T_a \bigoplus T_b|/3}{\text{テストケース数}} \tag{8}$$

- T_a, T_b: プログラム a, b に各テストケースを実行した結果の集合
- $A \cap B$: A と B の双方が通過したテストケースの集合
- $A \bigoplus B$: A と B の片方のみが通過したテストケースの集合
- $|A|$: A の要素数

4.1.3 双方での評価

先に説明した，類似度での評価法とテストケースでの評価法は互いに独立しているため，これらを組合わせて個体間評価に用いることが可能である．本研究における評価実験の章では，各評価法を単一で用いたパターンに加えて，2 つの評価方法を組合わせたパターンも実施している．組合わせは式 (9) に示すように双方を掛け合わせた値を使用している．この評価値の範囲も前者同様に [0..1] を取る．

$$\text{双方評価値} = \text{類似度評価値} \times \text{テストケース評価値} \tag{9}$$

4.2 交叉対象の選択

4.2.1 評価値を用いた選択

個体間評価値の計算を終えれば，次に交叉を行う組合わせを決定する．通常の状態では評価値が高い組合わせから順に一定回数選ぶ．GenProg では実行時設定で与えられた交叉確率に従って交叉処理を行うことで過剰な個体増加を防いでいる．このため提案手法でも式

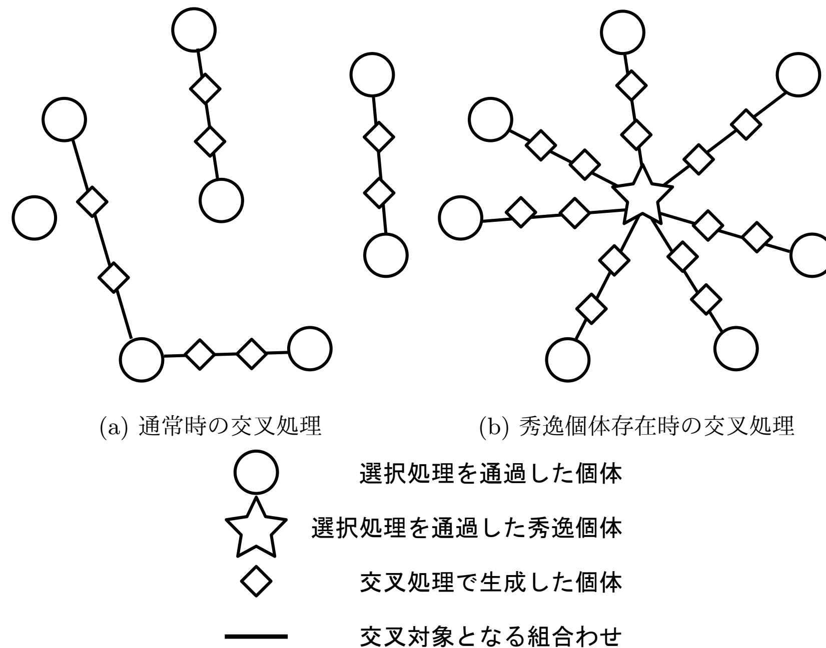


図 5: 秀逸個体の有無による交叉処理の変化

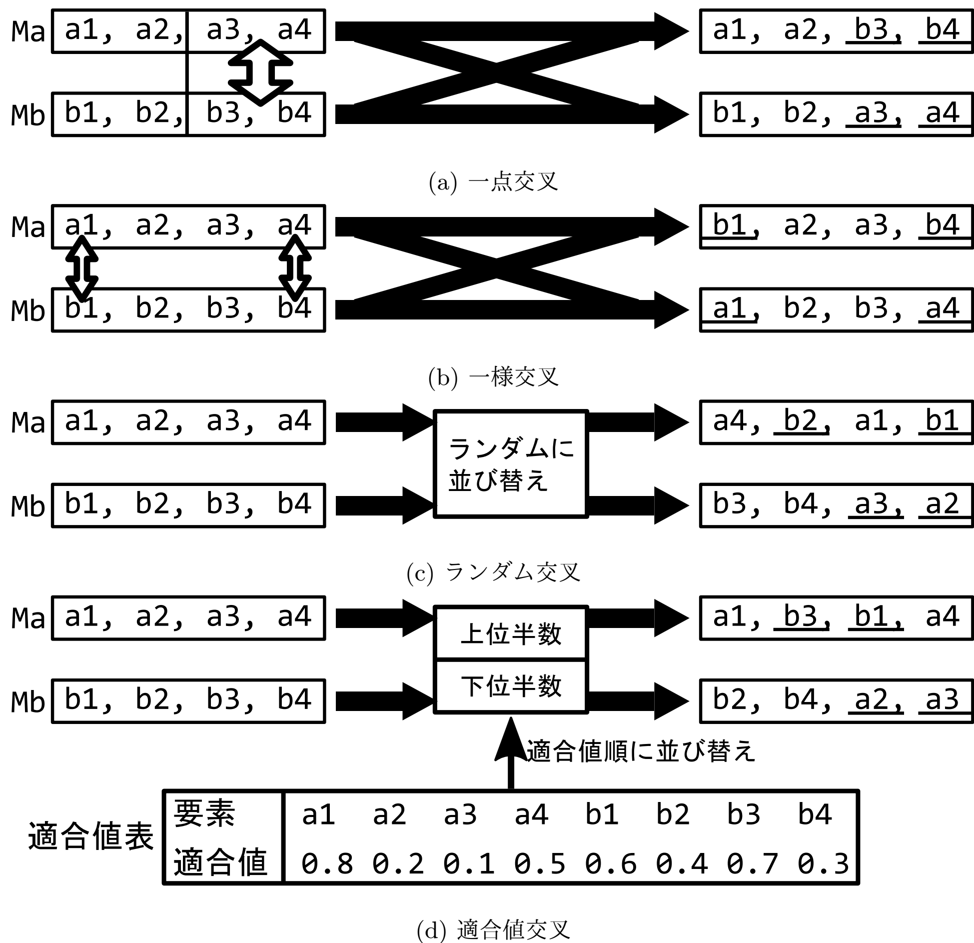
(10) で示される回数だけ交叉処理を行う．また，図 3(b) における個体 B が該当するように，同じ個体が複数回選択される可能性もある．

$$交叉回数 = \frac{修正プログラム候補の数 \times 交叉確率}{2} \tag{10}$$

4.2.2 秀逸個体を中心にした選択

生成した修正プログラム候補の中に適合値が極めて高いものが含まれる場合，そのプログラムに微調整を加えることで全テストを通過する修正プログラムが生成できる可能性が高い．そこで，提案手法では選択処理において適合値が最大の 2/3 を超えるものを**秀逸個体**と呼び，選択処理において秀逸個体が確認された場合は個体間評価値の計算を行わず，秀逸個体と他の全修正プログラム候補の組合わせで交叉処理を実行して修正プログラムの生成を狙う．

秀逸個体の有無による交叉処理の変化を図 5 に示す．通常時は 5(a) に示すように，個体間の評価値に応じて様々な組合わせで交叉を実行するが，秀逸個体が現れた場合は図 5(b) に示すように，秀逸個体と他の全個体との組合わせで交叉を実行する．通常時の交叉確率にもよるが，この場合は修正プログラム候補の数が大きく増加することが多いため，一世代辺りの処理時間もまた増加すると考えられる．したがって評価実験においては，全交叉を行うか否かを変えたパターンについても実行している．



X:入換わった要素　X:入換わらなかった要素

図 6: 交叉方法

4.3　交叉の実行

提案手法で使用する交叉パターンを図 6 に示す．GenProg では図 6(a) に示す一点交叉を使用しているが，提案手法ではこれに加えて図 6(b) に示す一様交叉 および図 6(c) に示すランダム交叉，図 6(d) に示す適合値交叉を使用する．以下，各交叉方式について説明する．

4.3.1　一点交叉

一点交叉とは，要素の集合を中間で区切り，後半の要素を入換え交叉方法である．この方法は極めて簡単なアルゴリズムであり，自動修正に限らず多くの遺伝的プログラミングで用いることが可能であるが一般的に性能が低いといわれる．特に，隣接する要素が最適解の生

成を妨げる**ヒッチハイキング**[1] に弱いとされる [23]．一点交叉に関連する交叉方法には二点交叉が存在し，これは要素の集合を二箇所で区切り，中間の領域を入換える．同様にして区切りを増やした交差方法は多点交叉や n 点交叉などと呼ばれる．区切りが多いほどヒッチハイキングへの耐性は向上するが，提案手法においては個体ごとの要素数が一定でないため，多点交叉は使用していない．

4.3.2　一様交叉

一様交叉とは，個体間で向かい合う要素をそれぞれ半分の確率で入換える交叉方法である．この方法は期待値で半分の要素を入換えることが可能であり，隣り合う要素が同時に入換えられる確率は 50%のため，ヒッチハイキングに耐性があるといえる．また，個体間で要素数が異なる場合，余った要素は半分の確率で交叉相手側に移すのみとする．

4.3.3　ランダム交叉

ランダム交叉とは，互いの個体が持つ全要素を一度まとめてからランダムに並び替え，要素数が均等になるように再分配する交叉方法である．また，この交叉方法を実行する場合は大半の要素の順番が変更されるため，一般的な遺伝的プログラミングに採用することは難しいが，提案手法においては要素の位置が実行結果に影響することはないため，手法性能に悪影響を及ぼすこともない．

4.3.4　適合値交叉

適合値交叉とは，全要素を一度まとめて適合値順に並び替えて上位と下位に分けて分配する交叉方法である．本手法における要素の適合値の定義は “その要素を持つ個体の適合値の平均” とする．したがって，優れた個体に多く現れる要素ほど高い適合値を持つことになる．記録した要素の適合値は世代を経ても保持し続け，新たな個体が生成されるたびに更新する．この交叉方法を用いる場合，生成される二個体の内一方は優れた要素を持ち，もう一方は劣る要素を持つことになる．したがって，他の交叉方法と異なり，生成される個体が平等でないという特徴を持つ．

[1]図 6 において要素 a1 が修正に必要で a2 が新たなバグを出す場合，交叉によって a1 と a2 を切り離す必要がある

5 評価実験

提案手法の有効性を示すため，提案手法を実装したツール Marriagent を作成し，GenProg との比較実験を行った．評価項目は修正の成功率と修正に要する時間，修正成功時の世代数である．

5.1 実験対象

実験対象には航空機制御システムである tcas[24] を選択し，3 つのバージョンを用意した．各バージョンの詳細を表 1 に示す．バージョン 9 は代入文に欠陥を含む．代入文の欠陥は制御構造に影響を与えにくいため，実行文の操作を行う提案手法で容易に修正できる．バージョン 43 は制御文に欠陥を含む．制御文の欠陥は制御構造に影響するものが大半であるため，提案手法による修正は代入文の修正に比べて困難である．バージョン 44 は他バージョンと異なり二箇所に欠陥を持っており，一箇所の変形では修正不可能なため修正が非常に困難である．

5.2 実験方法

前節で示した実験対象に対して，GenProg と Marriagent を 50 回ずつ実行し，修正が 15 分以内に成功した回数を記録する．同時に，修正成功時の実行時間および世代数も記録する．

また，実行によっては交叉を行う前，修正プログラム候補の初期生成の時点で修正に成功する場合がある．この場合は一度も交叉処理が行われず既存手法と提案手法の比較が無意味なため，これらを除外した結果を記録する．

表 1: 実験対象 tcas の詳細

バージョン	行数	欠陥の数	欠陥箇所
バージョン 9	173	1	代入文
バージョン 43	175	1	制御文
バージョン 44	175	2	代入文と制御文

表 2: 実験の実行設定

項目	設定値	項目	設定値
成功テスト数	100	失敗テスト数	9
1 世代の個体数	10	交叉確率	0.5
乱数	実行毎に変更	世代更新	無制限

5.3 実験設定

GenProg および Marriagent を実行した際の設定を表 2 に示す．乱数は 1 から 50 までの範囲で実行毎に変更し，世代更新は 15 分の制限時間内ならば無制限に進行させる．また，表 2 に記載していない設定はデフォルトのものを指定した．加えて，Marriagent 固有の実行設

表 3: Marriagent の実行設定

パターン名	個体間評価方法	交叉方法	秀逸個体の有無
GenProg	なし (ランダムに選択)	一点交叉	なし
MarriagentSOC	類似度	一点交叉	なし
MarriagentSOQ			あり
MarriagentSUC		一様交叉	なし
MarriagentSUQ			あり
MarriagentSRC		ランダム交叉	なし
MarriagentSRQ			あり
MarriagentSFC		適合値交叉	なし
MarriagentSFQ			あり
MarriagentTOC	テストケース	一点交叉	なし
MarriagentTOQ			あり
MarriagentTUC		一様交叉	なし
MarriagentTUQ			あり
MarriagentTRC		ランダム交叉	なし
MarriagentTRQ			あり
MarriagentTFC		適合値交叉	なし
MarriagentTFQ			あり
MarriagentBOC	双方使用	一点交叉	なし
MarriagentBOQ			あり
MarriagentBUC		一様交叉	なし
MarriagentBUQ			あり
MarriagentBRC		ランダム交叉	なし
MarriagentBRQ			あり
MarriagentBFC		適合値交叉	なし
MarriagentBFQ			あり

定を表 3 に示す．各手法は節 4.1 と節 4.3 で示したものである．以降，Marriagent の各実行設定を定めたものを**実行パターン**と呼ぶ．

5.4 実験結果

実験の結果を表 4，表 5 表 6 にバージョン別に示す．節 5.2 で述べたように，各評価項目について，バージョン 9 およびバージョン 43 に対する実験では全てのパターンが少なくとも一度は修正に成功していたが，バージョン 44 に対する実験では一度も修正に成功しなかった実行パターンが存在した．このため，該当する実行パターンの平均実行時間及び平均修正世代数の欄は斜線を引いている．また，バージョン 9 およびバージョン 43 においては初期

表 4: tcas バージョン 9 の結果

ツール名	初期世代抜き成功回数	初期世代抜き平均実行時間	初期世代抜き平均修正世代数
GenProg	13	251.54	4.00
MarriagentSOC	22	199.26	2.68
MarriagentSOQ	24	221.35	2.92
MarriagentSEC	22	248.06	3.86
MarriagentSEQ	12	184.30	2.58
MarriagentSRC	22	213.17	2.91
MarriagentSRQ	21	274.18	3.67
MarriagentSFC	16	181.43	2.44
MarriagentSFQ	16	163.59	2.06
MarriagentTOC	20	218.07	3.00
MarriagentTOQ	18	249.68	3.56
MarriagentTEC	17	210.99	2.88
MarriagentTEQ	14	235.44	3.14
MarriagentTRC	29	248.36	3.28
MarriagentTRQ	27	272.62	3.59
MarriagentTFC	26	341.80	4.50
MarriagentTFQ	32	368.71	4.75
MarriagentBOC	15	203.18	2.73
MarriagentBOQ	19	221.97	3.00
MarriagentBEC	17	214.43	2.94
MarriagentBEQ	11	158.08	2.00
MarriagentBRC	23	194.69	2.61
MarriagentBRQ	21	258.00	3.48
MarriagentBFC	18	225.39	3.17
MarriagentBFQ	21	245.37	3.14

初期世代での修正は 5 回

世代で修正に成功した実行があるため，これらを抜いた結果を合わせて記載している．

実行時間または修正世代数の箱ひげ図を図 7 および図 8 に示す．箱の範囲は第一四分位数から第三四分位数までで，太線が中央値を示す．また，実行時間と修正世代数の実行パターン別分布図を図 9 および図 10 に示す．バージョン 44 は母数が少なく統計的に有意な比較ができないため省略している．

表 4 および表 5 より，バージョン 9, バージョン 43 に対する実験では Marriagent の大半の実行パターンが GenProg よりも多くの成功回数を記録していることが分かる．特に，適合値交叉を用いた実行パターンは高い成功回数を記録しており，続いてランダム交叉を用いた実行パターンが優れた結果を残している．一方，表 6 よりバージョン 44 では GenProg は

表 5: tcas バージョン 43 の結果

ツール名	初期世代抜き 成功回数	初期世代抜き 平均実行時間	初期世代抜き 平均修正世代数
GenProg	4	234.92	3.75
MarriagentSOC	9	183.24	2.44
MarriagentSOQ	8	273.08	2.63
MarriagentSEC	5	128.60	1.80
MarriagentSEQ	9	190.65	2.00
MarriagentSRC	9	278.54	4.00
MarriagentSRQ	16	286.63	3.06
MarriagentSFC	12	352.77	5.00
MarriagentSFQ	9	281.54	2.89
MarriagentTOC	5	276.81	4.20
MarriagentTOQ	12	445.92	5.83
MarriagentTEC	5	274.30	3.80
MarriagentTEQ	8	183.29	2.13
MarriagentTRC	15	325.39	4.73
MarriagentTRQ	17	354.58	4.06
MarriagentTFC	17	373.43	5.29
MarriagentTFQ	12	408.15	4.83
MarriagentBOC	5	409.93	6.20
MarriagentBOQ	9	264.70	3.11
MarriagentBEC	5	141.16	2.00
MarriagentBEQ	8	152.61	1.63
MarriagentBRC	8	145.43	2.00
MarriagentBRQ	14	269.31	2.79
MarriagentBFC	7	248.53	3.57
MarriagentBFQ	8	368.23	4.38

初期世代での修正は 3 回

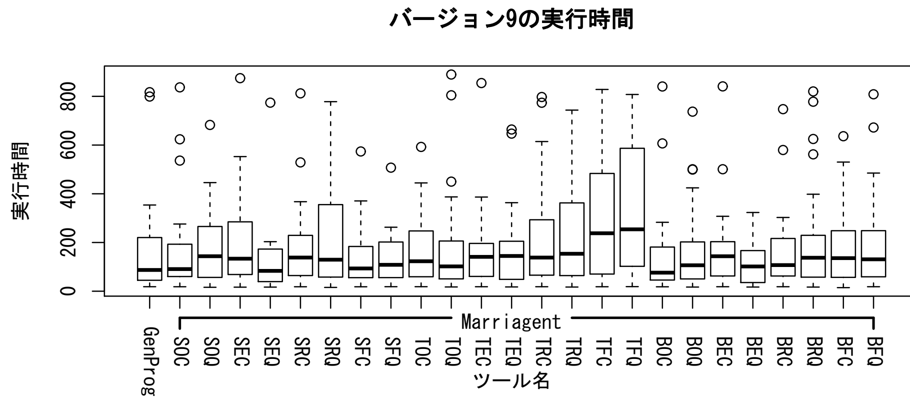
一度も修正に成功していないが，Marriagent では修正に成功したパターンが存在することが分かる．

実行時間および修正世代数については，図 7 および図 8 からどの実行パターンを用いた場合でも分散が大きいことに加えて，図 9 および 10 から実行時間と修正世代数の相関が強いことが分かる．バージョン 9 では一部を除けば実行パターンごとにあまり大きな差は生じていないが，一方でバージョン 43 では実行パターンによって大きく差が開いており，GenProg の半分程度のものもあれば二倍近いものも存在する．

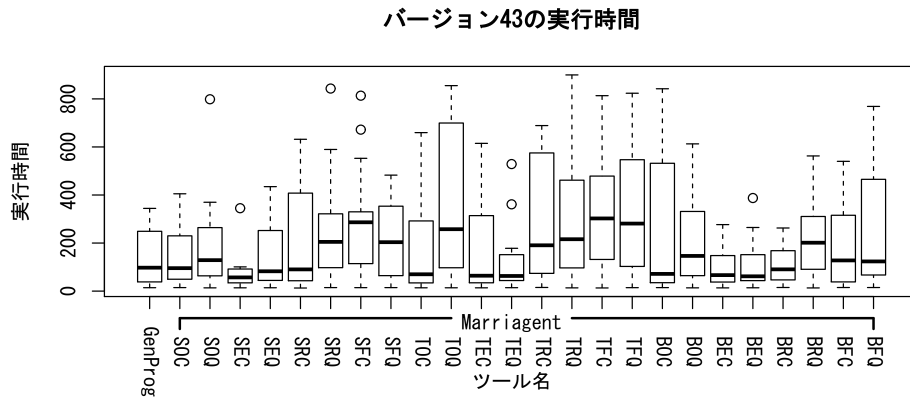
表 6: tcas バージョン 44 の結果

ツール名	成功回数	平均実行時間	平均修正世代数
GenProg	0		
MarriagentSOC	0		
MarriagentSOQ	3	338.30	4.00
MarriagentSEC	0		
MarriagentSEQ	0		
MarriagentSRC	0		
MarriagentSRQ	3	446.73	5.33
MarriagentSFC	1	133.14	2.00
MarriagentSFQ	4	590.30	6.25
MarriagentTOC	1	844.01	11.00
MarriagentTOQ	1	430.92	5.00
MarriagentTEC	0		
MarriagentTEQ	1	855.15	11.00
MarriagentTRC	3	377.42	4.67
MarriagentTRQ	4	638.91	7.75
MarriagentTFC	5	476.96	5.40
MarriagentTFQ	3	489.36	6.00
MarriagentBOC	0		
MarriagentBOQ	1	403.61	5.00
MarriagentBEC	0		
MarriagentBEQ	0		
MarriagentBRC	1	371.64	5.00
MarriagentBRQ	3	452.25	5.33
MarriagentBFC	1	106.58	1.00
MarriagentBFQ	2	411.25	5.00

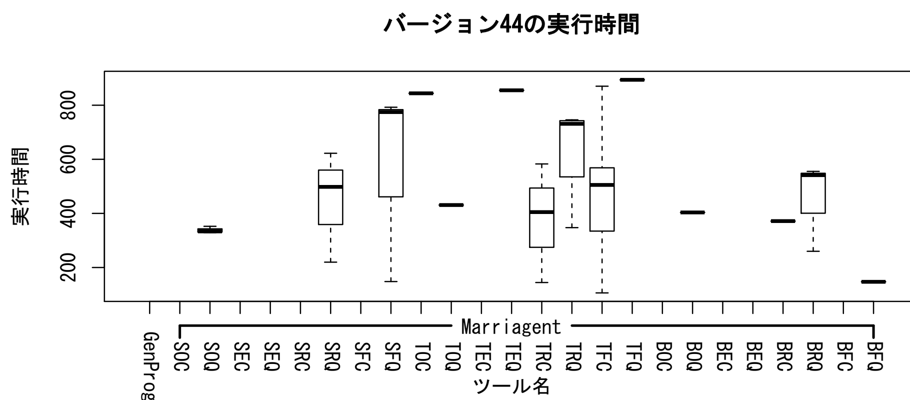
初期世代での修正は無し



(a) バージョン 9 の実行時間

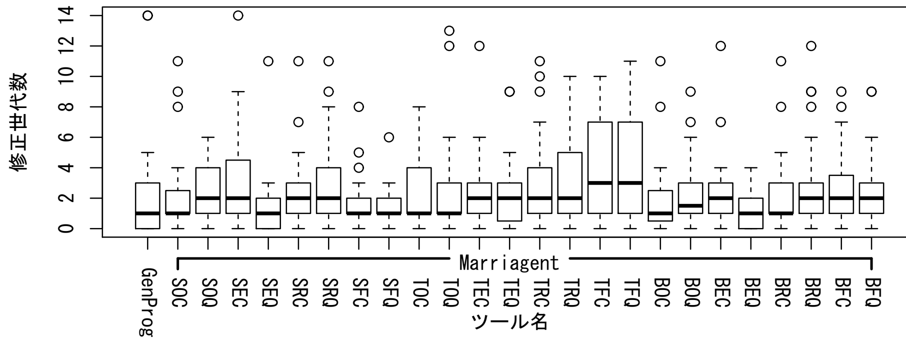


(b) バージョン 43 の実行時間

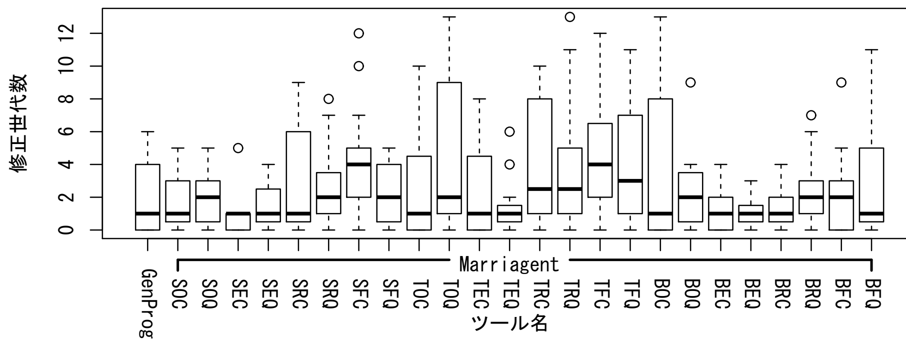


(c) バージョン 44 の実行時間

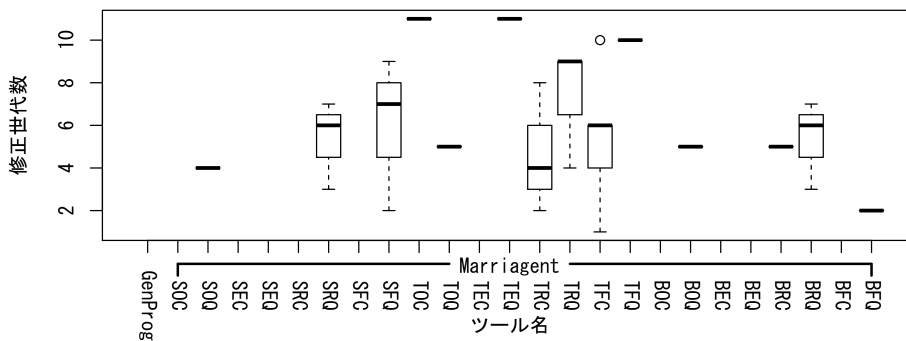
図 7: 実行時間



(a) バージョン 9 の修正世代数

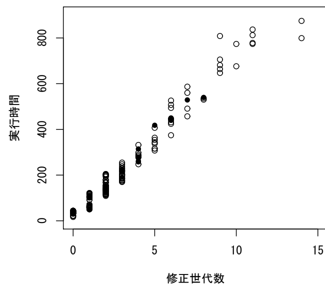


(b) バージョン 43 の修正世代数

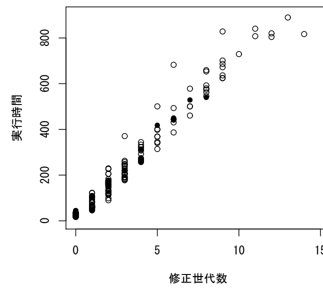


(c) バージョン 44 の修正世代数

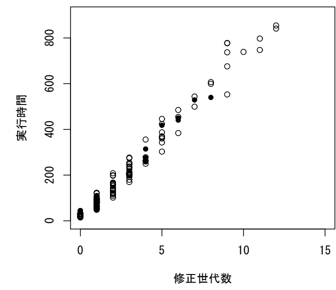
図 8: 修正世代数



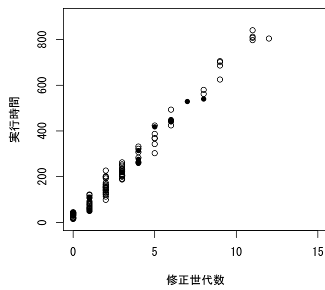
(a) 類似度による評価



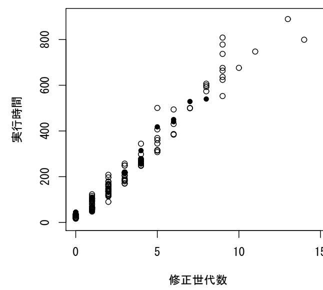
(b) テストケースによる評価



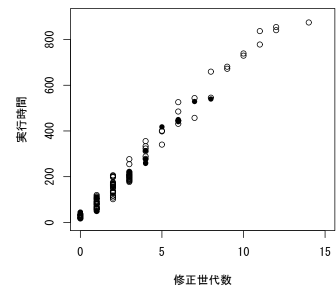
(c) 双方による評価



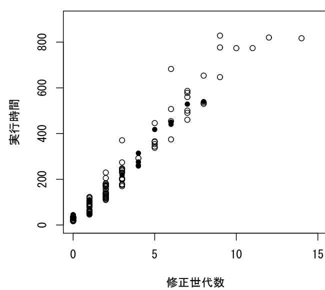
(d) 一点交叉



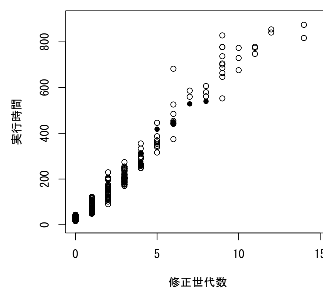
(e) 一様交叉



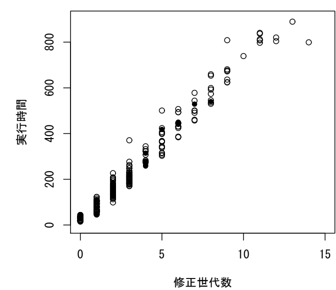
(f) ランダム交叉



(g) 適合値交叉



(h) 秀逸個体なし



(i) 秀逸個体あり

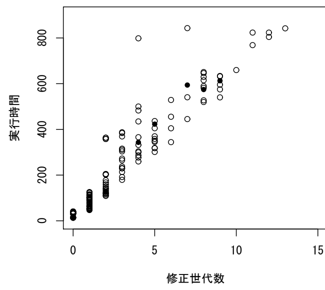
図 9: バージョン 9 における実行時間と修正世代数の分布図

6 考察

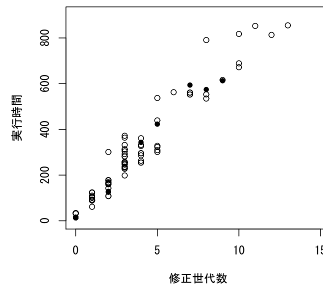
6.1 個体間評価方法

6.1.1 類似度での評価

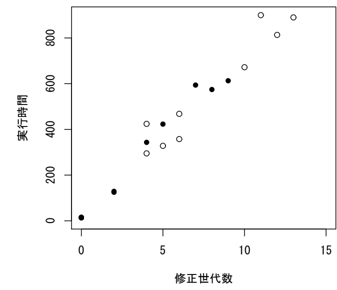
バージョン 9 およびバージョン 43 において，交叉対象を類似度で評価した実行パターンは交叉方式によらず優れた成功回数を記録している．このような結果が得られた理由は，本



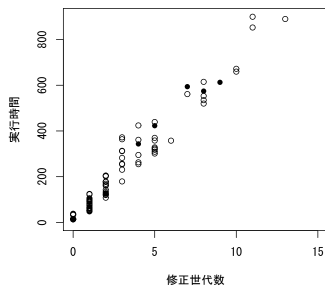
(a) 類似度による評価



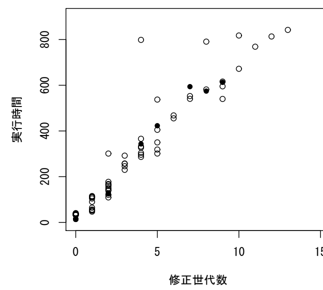
(b) テストケースによる評価



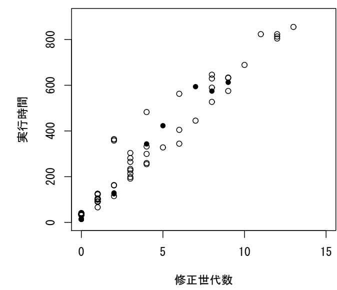
(c) 双方による評価



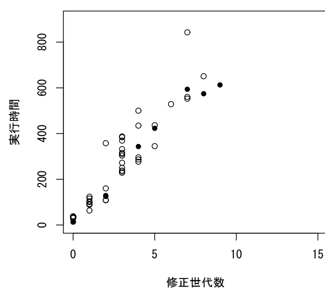
(d) 一点交叉



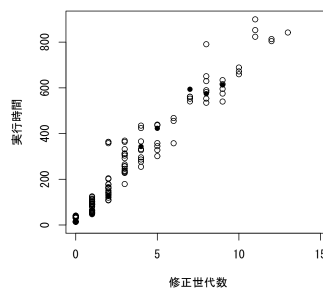
(e) 一様交叉



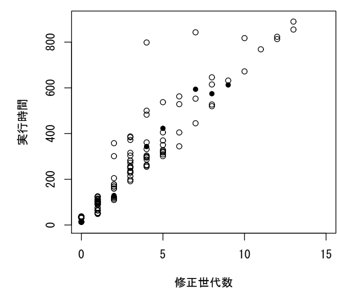
(f) ランダム交叉



(g) 適合値交叉



(h) 秀逸個体なし



(i) 秀逸個体あり

図 10: バージョン 43 における実行時間と修正世代数の分布図

研究の動機である交叉処理によって大きな変化を狙うという考えが効果的であったためだと考えられる．他の評価法に比べて交叉方式による成功回数の変化が控えめな理由は，共通要素が僅かまたは存在しない組合わせで交叉を行うため，どのような入換え方でも大きな変化を起こせることが大きいと考えられる．

実行時間および修正世代数はランダム交叉がやや長い程度で，全体では大きな差が生じていない．以上より，類似度での評価で交叉対象を選ぶ場合は他の実行設定にあまり依存し

ない．

6.1.2　テストケースでの評価

交叉対象をテストケースで評価したパターンのうち，交叉方式にランダム交叉または適合値交叉を使用した実行パターンは非常に優れた結果を出しているが，一方で一点交叉および一様交叉を使用した実行パターンは全体でやや劣る結果を出している．このような結果が得られた理由は，本評価法によって交叉対象を選択した場合は必然的に優れた個体から順番に選ばれるため，類似した個体の組合わせが起こりやすいことが主として考えられる．したがって，要素の順番が変更されない一点交叉および一様交叉を使用した場合は個体間で重複した要素が入換えられる可能性が高い．その場合，交叉前の個体と類似した個体，もしくは全く同じ個体が生成されることが多々起こりうると推測される．対照的に，ランダム交叉及び適合値交叉は要素の順番に依存しないため，類似した個体同士の組合わせでも大きく変化した個体を生成できる．

また，実行時間及び修正世代数は交叉方式によって非常に大きな差が生じており，一様交叉を用いた実行パターンは短時間で終わるものが多く，適合値交叉を用いた実行パターンは長時間を要するものが多い．ただし，両実行パターンの間では修正回数に大きな差が開いており，どちらも 5 世代以下における修正成功が大半を占めている。したがって，両実行パターンに生じた差は高世代における修正の有無によるものだと考えられる．以上より，類似個体で交叉処理を行う場合はできるだけ別の要素を入換える交差方式を用いるべきである．

6.1.3　双方での評価

類似度とテストケースの双方で評価した実行パターンは，各評価法を単体で評価した実行パターンに比べて若干劣る結果となった．この原因としては，双方の評価法の相性が悪く互いの長所を阻害している可能性が考えられる．遺伝的プログラミングは優れた個体を選び，それを元に新たな個体を生成するというアルゴリズムであるため，世代が進むにつれて同じ個体を基に生成された類似個体が多く生成されることが多い．特にテストケースによる評価法を用いて交叉対象を選択した場合は同一個体が複数回にわたって交叉対象に選ばれやすく，交叉処理を通して多数の類似個体が生成されることがある．この場合，類似個体の組合わせが増加するため類似度による評価法が効果を発揮できなくなると考えられる．ただし，実行時間は短い傾向にあるため，類似個体が少ない初期段階では双評価法を用いることで優れた修正性能を出せるといえる．

以上より，類似度による評価法とテストケースによる評価法はどちらも優れた点を持っているが，これらを組合わせて用いる場合，単純に組合わせるのではなく世代数や適合値に応

じて動的に切り替えたり，もしくは重み付けや閾値を設けたりするなどの工夫が必要だと考えられる．

6.2　交叉方式

6.2.1　一点交叉

一点交叉を用いた実行パターンはどの個体間評価法を用いた場合も GenProg より優れた結果を出しており，特に類似度評価を用いた場合は差が顕著であった．GenProg も一様交叉を使用しているため差が生じた要因は交叉対象の選択方法および秀逸個体の有無だが，言い換えればこれらを工夫することで一点交叉でも効果的な交叉を行えることが示された．また，提案手法における一点交叉は要素配列を中間で区切るが，世代が進むにつれて新たな要素が配列末尾に追加されるため区切りの位置も移動する．したがって，区切りを徐々に移動させる性質が個体の変化に好影響をもたらしたと考えられる．特に，節 4.3 で述べたヒッチハイキングへの耐性が得られたことも今回の結果が得られた要因として考えられる．

実行時間及び修正世代数について図 9(d) および図 10(d) から，ランダム交叉や適合値交叉に比べて相関が強いこと，すなわち 1 世代あたりの実行時間に分散が少ないことが分かる．これは一点交叉のアルゴリズムが簡潔であり，どの個体を対象に実行した場合でも計算時間がほとんど変化しないためだと考えられる．

6.2.2　一様交叉

一様交叉を用いた実行パターンは他の交叉方式を用いた実行パターンに比べて劣る結果となった．特に，バージョン 9 の実行結果で GenProg より少ない修正回数を記録した実行パターンはすべて一様交叉を用いたものであり，明らかに修正性能が劣るといえる．

ランダム性の高さおよびヒッチハイキングへの耐性はランダム交叉と同様のため，一様交叉が有効でない理由は変化の起こしにくさだと考えられる．ランダム交叉は要素の順番を入れ換えるため類似固体を対象にした交叉でも変化を起こしうるが，一様交叉は向かい合う要素を入れ換えるのみで大きな変化を起こしにくい．以上より，提案手法においては一様交叉の有効性は低いといえる．

6.2.3　ランダム交叉

ランダム交叉を用いた実行パターンは高い成功回数を記録しており，特に個体間評価法にテストケースを用いた場合は非常に高い結果を記録している．このような結果が得られた理由は上述のように変化量が大きいことが考えられ，特にテストケースを用いて選ばれた類似した個体に対しても十分な変化を与えられる点が大きい．

一方で，元々類似した個体が選ばれにくい類似度評価を用いた場合は一点交叉をさほど変わらない結果を出している．秀逸個体を用いる場合は大きな差が生じているが，この場合は類似固体同士の交叉が多数行われ得るため，ランダム交叉は効力を発揮すると考えられる．

6.2.4 適合値交叉

実験結果において最も優れた結果を残した実行パターンはいずれも適合値交叉を用いたものであった．このような結果が得られた理由は優れた要素のみを集め，不要な要素を除去するという考えが効果を発揮したためだと考えられる．言い換えれば，変異処理によって加えられた要素が良いものか悪いものかを判別することで有利に交叉処理を行える．今回の設定では計算時間を考慮して個体ごとの平均評価値を要素の評価値と定義したが，個体の性質に応じて重みづけを行うなどの工夫をすることで更なる修正性能の向上が期待できる．

また，図 9(g) および図 10(g) に示されるようにこの交叉方式を用いた実行パターンは世代数に比べて実行時間が長くなる傾向にある．これは他の交叉方式に比べて要素間比較に要する時間が加わるためであり，計算のアルゴリズムを改良することでも手法の有効性が向上すると考えられる．

6.3 秀逸個体の有無

バージョン 9 においては秀逸個体を用いる実行パターンは秀逸個体を用いない実行パターンに比べて若干劣る結果を出し，一方でバージョン 43 およびバージョン 44 においては秀逸個体を用いる実行パターンの方が優れた結果を出した．バージョン間で異なる結果が得られた理由としては，元々の修正難易度，すなわち解への収束しやすさが考えられる．秀逸個体を用いる場合は個体間評価を行わず全ての他個体を交叉対象とするため，これによって優れた組合わせが得られる場合もあれば悪い組合わせが得られる場合もある．したがって，通常の交叉対象の選択で優れた組合わせが得やすいバージョン 9 では悪い組合わせの増加が悪影響を及ぼし，悪い組合わせが得やすいバージョン 43 およびバージョン 44 では良い組合わせの増加が好影響を及ぼすと推測される．

また，図 9(i) および図 10(i) から読み取れるように，秀逸個体が現れる世代は個体数が大幅に増加するため，状況によっては計算時間の浪費に繋がり結果に悪影響を及ぼす可能性がある．以上より，秀逸個体を中心とした交叉を用いるか否かは秀逸個体の有無のみでなく，世代数や平均適合値などから修正難易度を推測して行うべきである．

7 妥当性の脅威

本研究では評価実験として GenProg と Marriagent の比較を行ったが，GenProg の実行設定は表 2 で設定した以外に変異の確率やテストケースの重み付けなど，多岐にわたる項目が存在する．このため，その他項目の設定を変更することで本実験とは大きく異なった結果が得られる可能性がある．特に提案手法は個体の組合わせに強く依存するため，1 世代辺りの個体数や世代数の上限など個体の生成に関する項目を変更した場合，大きく結果が変わる恐れがある．

また，今回実験対象としたソフトウェアである tcas の規模は高々百数十行であるため，数千行もしくは数万行の大規模なソフトウェアを実験対象に用いた場合は手法の有効性，特に類似度を用いた評価法の有効性が変化する可能性がある．また，テストケースの数も合計 109 個と多数使用しているためテストケースを用いた評価が有効であったが，高々数個のテストケースを用いた場合はプログラム毎のテストケース通過数に差が開きにくくなるため，ソフトウェア規模と同様に手法の有効性に影響すると考えられる．このような，ソフトウェアを変更した際に生じる提案手法の有効性の変化の調査は今後の課題となる．

8　あとがき

本研究では，GenProg の交叉処理を改良し，個体すなわち修正プログラム候補の中からから効果的な組合わせを選択して交叉を行う手法を提案した．実験の結果より，テストケース通過の是非をもって交叉対象を選択し，優れた個体に現れる個体を片方に集めるように交叉処理を行うことで，高い確率で修正プログラムを生成できることが示された．今後の課題としては，個体間評価方法および交叉方式について新たなアルゴリズムの考案および既存のアルゴリズムとの組合わせの他，選択や変異に関する他の手法に提案手法を組合わせた場合の修正性能の評価，大規模ソフトウェアをはじめとした実験対象の拡大が挙げられる．

謝辞

本研究を行うにあたってご理解とご指導を賜り，日頃より大いな励ましを頂きました，楠本 真二教授に深く感謝申し上げます.

本研究の全過程を通してご指摘およびご助言を頂き，常にご協力を頂きました肥後 芳樹准教授に心よりの感謝を申し上げます.

本研究に関して多くのご助言を頂き，様々なご協力をして頂いた柗本 真佑助教に深く感謝いたします.

本研究に加えて様々な場面でご協力をして頂いた楠本研究室の皆様に厚く御礼申し上げます.

最後に本研究を行うに至るまで，講義や実験等の活動を通して多大な知識や経験を頂きました，大阪大学大学院情報科学研究科の諸先生方および事務員方に，この場を借りて心からの御礼を申し上げます.

参考文献

[1] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, Vol. 7, No. 2, pp. 87–90, Apr. 2009.

[2] J. Baker. Experts battle ￡192bn loss to computer bugs, Feb. 2012. Accessed:2015-12-07, http://www.cambridge-news.co.uk/Experts-battle192bn-loss-bugs/story-22514741-detail/story.html.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *In 28th International Conference on Software Engineering (ICSE) 2006.*, pp. 361–370, May 2006.

[4] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *In 2006 International Workshop on Mining Software Repositories (MSR) 2006*, pp. 173–174, May 2006.

[5] T.H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *In 11th Working Conference on Mining Software Repositories (MSR) 2014*, pp. 82–91, May 2014.

[6] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *In 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE) 2011*, pp. 416–419, Sept. 2011.

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

[8] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *In 20th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2005*, pp. 273–282, Nov. 2005.

[9] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *In 30th International Conference on Software Maintenance and Evolution (ICSME) 2014*, pp. 191–200, Sept. 2014.

[10] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. Mimic: Locating and understanding bugs by analyzing mimicked executions. In *In 29th ACM/IEEE Inter-*

national Conference on Automated Software Engineering (ASE) 2014, pp. 815–826, Sept. 2014.

[11] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. Systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *In 34th International Conference on Software Engineering (ICSE) 2012.*, pp. 3–13, June 2012.

[12] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *In 36th International Conference on Software Engineering (ICSE) 2014.*, pp. 254–265, June 2014.

[13] D.Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *In 35th International Conference on Software Engineering (ICSE) 2013.*, pp. 802–811, May 2013.

[14] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *In 35th International Conference on Software Engineering (ICSE) 2013.*, pp. 772–781, May 2013.

[15] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *In 37th International Conference on Software Engineering (ICSE) 2015.*, pp. 448–458, May 2015.

[16] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. Van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, Vol. 82, No. 11, pp. 1780–1792, nov, 2009.

[17] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection.* MIT Press, Dec. 1992.

[18] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *In 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) 2014*, FSE 2014, pp. 306–317, Nov. 2014.

[19] M. 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, June 2014.

[20] L. Fan and R. Martin. Staged program repair with condition synthesis. In *In 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE) 2015*, pp. 166–178, sep. 2015.

[21] L. Fan and R. Martin. Automatic patch generation by learning correct code. In *In 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) 2016*, Jan. 2016. (In appear).

[22] XBD. Le, TDB. Le, D. Lo, and C. Le Goues. History driven automated program repair. In *In 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016.*, mar. 2016. (In appear).

[23] S. Forrest and M. Mitchell. Relative building block fitness and the building block hypothesis. In D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pp. 109–126. Morgan Kaufmann, Feb. 1993.

[24] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, Vol. 10, No. 4, pp. 405–435, oct. 2005.