非同期進化的アルゴリズムによるプログラム進化

原田 智広

電気通信大学 大学院情報理工学研究科
博士 (工学) の学位申請論文

2015 年 3 月

非同期進化的アルゴリズムによるプログラム進化

論文審査委員会

主査　髙玉　圭樹　　教授
委員　吉浦　裕　　教授
委員　西野　哲朗　　教授
委員　高橋　治久　　教授
委員　高橋　裕樹　准教授

Copyright (C) 2015 Tomohiro HARADA All Rights Reserved.

Abstract

This thesis focuses on Evolutionary Algorithm (EA) as one of meta-heuristic optimization approaches that explore an optimal solution not depending on properties or prior knowledge of given problems, proposes a novel asynchronous EA that asynchronously evolves solutions without waiting for evaluations of all solutions, and aims at investigating the effectiveness of the proposed asynchronous EAs. Concretely, this thesis proposes (1) Tierra-based Asynchronous EA (TAEA) that evolves solutions according to the *absolute* evaluation, employing the essential mechanism of the biological evolution simulator, Tierra, where digital creatures are asynchronously evolved, and adaptive TAEA (TAEA$^+$) that introduces the mechanism to adjust fitness values during evolution process; and (2) EA using Asynchronous Reference-based Evaluation (ARE-EA) that evolves solutions according to the *relative* evaluation which is asynchronously updated by evaluation of partial solutions. To investigate the effectiveness of the proposed asynchronous EAs, this study conducts experiments on the following three problems: (1) the symbolic regression problems as well known GP testbeds; (2) the assembly language programs evolution problems for practical applications; and (3) the assembly language programs evolution problems under the environment where bit inversions of programs occur for an application of spacecraft.

The intensive experiments by applying Genetic Programming (GP) as one of well-known EA into the proposed methods, TAGP (including TAGP$^+$) and ARE-GP, have revealed the following implications: (i) the proposed asynchronous EAs have the same or better regression ability than the conventional synchronous EAs even though the proposed EAs evolve the solutions according to only evaluations of *partial* solutions unlike the conventional ones evolve the solutions according to evaluations of *all* solutions; (ii) the proposed asynchronous EAs have a possibility to improve its regression ability when the evaluation times of the solutions differ from each other; (iii) the proposed asynchronous EAs cannot only maintain the correct programs, but also can evolve them to minimize their execution steps; (iv) the proposed asynchronous

EAs can evolve the programs even though the programs change and variable error of programs occurs due to bit inversions. The concrete result of the above four implications are summarized as follows: (i) in the symbolic regression problems, TAGP$^+$ and ARE-GP have better regression ability than $(\mu + \lambda)$-GP as a synchronous EA and has the same or better regression ability than ASSGP as an asynchronous EA; (ii) ARE-GP has a possibility to improve its regression ability in a case where evaluation times of the solutions differ from each other; (iii) in the assembly language programs evolution problems, TAGP, ARE-GP, and steady-state GP (SSGP), which is a conventional synchronous GP, can maintain the correct programs. In particular, TAGP and ARE-GP can evolve programs that have shorter execution steps than ones evolved by SSGP; and (iv) in the environment where bit inversions of programs occur, TAGP and ARE-GP cannot only maintain the correct programs, but also evolve the programs that has shorter program size than initial programs.

概要

本研究では，問題の特性や事前知識によらず汎用的に最適解を獲得可能なメタヒューリスティック最適化手法の一つである進化的アルゴリズム（Evolutionary Algorithm：EA）に着目し，解を同期的に進化させることが困難な問題に対して，すべての解の評価を待たずに非同期に進化させる非同期進化法を提案し，その有効性を検証することを目的とする．その目的達成に向けて本研究では，二種類の非同期 EA を提案する．一つ目は，非同期なデジタル生物の進化をシミュレートする Tierra を拡張し，絶対評価にもとづいて解を進化させる Tierra 型非同期 EA（Tierra-based Asynchronous EA：TAEA）を提案するとともに，適合度を進化の過程で適応的に調節する機構を導入した適応型 TAEA（TAEA$^+$）を提案する．二つ目は，非同期に得られる部分的な解評価から更新される指標を用いて相対評価に基づいて解を進化させる非同期リファレンス評価を用いる EA（Asynchronous Reference-based Evaluation EA：ARE-EA）を提案する．

提案手法の有効性を検証するために，本研究では (1) 一般的なベンチマーク問題である関数同定問題における関数進化，(2) 実応用を想定したアセンブリプログラム進化，(3) 宇宙機への適用を念頭においたプログラムのビット反転が生じる環境でのアセンブリプログラム進化を実験する．

提案手法を EA の代表的な手法である遺伝的プログラミング（Genetic Programming：GP）に適用した結果，提案非同期進化法である TAGP（TAGP$^+$ を含む）と ARE-GP は，(i) 部分的な解の評価のみで，全体の解の評価を利用可能な従来の同期進化法と同等以上の関数近似性能を達成可能であり，(ii) 解の評価時間のばらつきが大きいほど関数近似性能が向上する．また，(iii) 与えられた正常なプログラムを維持可能であることに加えて，実行ステップ数の最小化が可能である．さらに，(iv) プログラムの改変や変数のエラーが発生してもプログラム進化を可能にすることを明らかにした．具体的には，(i) に関しては，関数同定問題において，TAGP$^+$ と ARE-GP は同期 GP の $(\mu + \lambda)$-GP を上回る関数近似性能を示し，非同期 GP である非同期 steady-state GP（ASSGP）と比較しても同等以上の性能を示すことが明らかになった．(ii) に関しては，評価時間にばらつきがある場合に評価時間にばらつきがない場合と比べて ARE-GP の関数近似性能が向上する可能性があることが示唆された．(iii) に関しては，アセンブリプログラムの進化にお

いて，TAGP と ARE-GP，同期 GP である SSGP はいずれも与えられた正常なプログラムを維持可能であり，その中でも TAGP と ARE-GP は SSGP では獲得できない実行ステップ数の短いプログラムを生成可能であることが分かった，(iv) に関しては，プログラムのビット反転が発生する環境において，TAGP と ARE-GP は正常なプログラムを維持可能であり，さらに初期に与えたプログラムよりもサイズの小さなプログラムを生成可能である．

目次

図目次

表目次

第 1 章

序論

1.1 研究背景

近年，実社会における様々な最適化問題の解法として，生物の進化のメカニズムをもとにした多点探索による進化的アルゴリズム（Evolutionary Algorithm：EA）が注目を集めている．EA は問題の特性や事前知識によらず汎用的に（準）最適解を獲得可能なメタヒューリスティックの一種であり，代表的な手法としては，遺伝的アルゴリズム（Genetic Algorithm：GA）[2]，遺伝的プログラミング（Genetic Programming：GP）[3]，進化戦略（Evolutionary Strategy：ES）[4]，粒子群最適化（Particle Swarm Optimization：PSO）[5]，差分進化（Differential Evolution：DE）[6] などが挙げられる．また，近年では単一の評価関数だけではなく，複数の評価関数を同時に最適化する多目的最適化（Multi-Objective Optimization：MOO）に EA を適用した多目的進化的アルゴリズム（Multi-Objective Evolutionary Algorithm：MOEA）も注目を集めており，代表的な方法として NSGA-II（Non-dominated Sorting GA-II）[7] や MOEA/D（MOEA on decomposition）[8] などが提案されている．

このような EA は，多数の最適解候補を生成して解探索する多点探索を基本としており，(1) 初期の解候補の生成，(2) 評価関数に基づく解候補の評価，(3) 評価に基づく新しい探索点の生成のプロセスを繰り返すことによって解を進化させ，最適解を獲得する．EA の大きな特徴として，問題の事前知識によらず評価関数と遺伝子の設計のみによって適用可能である点，関数最大化などの実数関数の最適化だけでなく，巡回セールスマン問題やナップサック問題に代表されるような組合せ最適化にも適用可能であり，適用範囲が広い点が挙げられる．また，このような特徴から EA は実問題の最適化にも適用されている．例えば，新幹線 N700 系のフロントノーズの形状最適化 [9] や航空機 MRJ の翼の形状最適化 [10] のような工学設計や，Web 記事要約サービスを提供するスマートフォンアプリケーションの Summly[11] における要約語の組合せ最適化にも EA が適用されている．また，EA の各手法では一般に多数の解評価を繰り返すことで最適化するため，解評

価をいかに効率的に実行するかが重要になり，近年の並列計算技術の発展に伴い，EA の分野においても多数の解を並列に評価し，効率的な解探索を実現する並列型 EA（Parallel EA：PEA）が多数提案されている [12, 13, 14, 15]．これらの PEA では，各並列計算機に解の評価，あるいは解集団を分散することで解評価の効率化を目指している．例えば，[12] は GA，[13] は DE，[14] は PSO，[15] は MOEA/D の解評価を並列計算機によって実行する方法を提案している．

しかし，EA では世代（generation）に基づいて個体を進化させるため，母集団内の全個体の評価が同時に得られる必要がある．そのため，並列に評価をした場合には最も評価時間の遅い個体の評価が完了してからでないと次世代の母集団を生成できず，個体の評価時間が均一でない場合に計算時間の無駄が大きくなるという問題がある．特に，並列計算環境において個体の評価時間が均一でない状況は容易に起こりえるため，解決すべき課題である．例えば，(1) 個体の性質によって評価時間が異なる場合や，(2) 並列計算環境において計算機の性能が異なる場合などである．さらに，扱う問題によっては評価時間が長くなるだけではなく，個体の評価自体が完了しないという状況も考えられる．例えば，並列計算環境において計算機間の通信エラーが発生する場合や，GP において個体として進化させるコンピュータプログラムに無限ループが含まれる場合などである．このように個体の評価が得られない場合，従来の EA においては，評価の待機時間に上限を設け，その時間を経過しても個体が得られない場合には評価不能として打ち切り，適合度に最低値を与える方法や解を再評価する方法が一般的である [16]．しかし，解の性質が未知の問題においては適切な待機時間を設定することが困難であり，待機時間の設定が解探索性能に大きな影響を与える．待機時間が長すぎる場合には計算時間が無駄になり，逆に短すぎる場合には評価が完了するはずの個体を無視することになる．そのため，従来の EA ではこれらの状況への対処が困難である．

これに対し，近年では全個体の評価を待たずに非同期（*asynchronous*）に個体を進化させる方法（非同期 steady-state GP[17]，非同期 PSO[18, 19]，Jinetic[20]，NEX[21]，多目的非同期 PSO[16] など）が提案されている．非同期進化では，各個体を独立に進化させるため，評価時間の長い個体を待たずに進化を継続でき，計算時間を無駄にせず効率的な解探索が可能になる．しかし，これらの手法 [17, 18, 19, 20, 21, 16] は，評価の完了しない個体に対しては従来の同期進化と同様に待機時間の上限を与えることで対処しており，同期進化と同様の問題を抱えている．

1.2　研究目的と方法

上記の問題を解決するために，本研究では (1) 解を非同期に進化可能であり，かつ，(2) 評価が完了しない解に対しても待機時間の上限を設けずに対処可能な新しい非同期進化的

アルゴリズムを提案し，その有効性を検証することを目的とする．具体的には，本研究では二種類の非同期 EA を提案する．一つ目は，非同期なデジタル生物の進化をシミュレートする Tierra[22] を拡張し，絶対評価にもとづいて解を進化させる Tierra 型非同期 EA（Tierra-based Asynchronous EA：TAEA）を提案する．TAEA では，あらかじめ定めた絶対的な評価に基づいて部分情報だけでは判断が難しい解の優劣を判断することで非同期な進化を可能にする．二つ目は，非同期に得られる部分的な解評価から更新される評価指標を用いて相対評価に基づいて解を進化させる非同期リファレンス評価を用いる EA（Asynchronous Reference-based Evaluation EA：ARE-EA）を提案する．ARE-EA では，部分的に得られる優良解の評価に基づいて，解の優劣を決定する相対的な評価指標を更新し，その指標に基づいて非同期な進化を可能にする．ここで，上記の 2 つの手法はプログラム進化を扱う GP をもとに展開する．その理由は，(1) プログラムはプログラムサイズや構造によって評価（実行）時間に差が生じる，(2) 無限ループによって評価が完了しない個体が生成される可能性があり，非同期進化が適しているためである．

提案手法の有効性を検証するために，TAEA，および ARE-EA を次の問題に適用し，従来の EA との性能を比較する．具体的には，(1)GP の一般的なベンチマーク問題である関数同定問題（symbolic regression problem）[3]，(2) 実応用を想定した PIC マイコンのアセンブリプログラム進化，および (3) 宇宙機への適用を念頭においた宇宙放射線によるプログラムのビット反転が発生する環境下でのアセンブリプログラム進化の三種類の例題に適用する．

1.3 本論文の構成

本論文の構成は以下の通りである．

まず，第 2 章では本研究で扱う EA について概観する．はじめに，EA の代表的な方法である同期 EA として，遺伝的アルゴリズム（Genetic Algorithm：GA），プログラムの進化を扱う遺伝的プログラミング（Genetic Programming：GP）についてそれぞれ説明する．続いて，EA を並列計算環境で実行する並列型 EA（Parallel EA：PEA）を紹介する．

第 3 章では，従来の EA において解の評価時間にばらつきがある場合の問題点について述べ，それに対するアプローチとして解を非同期に進化させる非同期 EA として差分進化（Differential Evolution：DE），粒子群最適化（Particle Swarm Optimization：PSO），MOEA/D（MOEA on decomposition），非同期粒子群最適化（asynchronous PSO；APSO），非同期 steady-state GP（asynchronous steady-state GP：ASSGP）について説明する．続いて，従来の非同期 EA についてその非同期性を分類し，本研究の位置づけを明らかにする．

第 4 章では，一つ目の提案手法である Tierra 型非同期進化的アルゴリズム（Tierra-based Asynchronous EA：TAEA）について説明する．具体的には，TAEA の基となるデジタル生物の進化シミュレータ Tierra について説明する．続いて，Tierra からの改良点および詳細な TAEA アルゴリズムについて述べ，TAEA の非同期進化を促進する選択法として TD トーナメント選択（temporal difference tournament selection：TDTS）を提案する．最後に，TAEA に適合度関数の自動調整と個体の削除パラメータを自動的に調節する手法を加えて拡張した適応型 TAEA（TAEA$^+$）を提案する．

第 5 章では，TAEA の問題点を解決した方法として非同期リファレンス評価を用いる進化的アルゴリズム（Asynchronous Reference-based Evaluation EA: ARE-EA）について説明する．具体的には，TAEA の問題点を述べ，それに対する解決策と ARE-EA の設計方針を示す．続いて，ARE-EA の詳細なアルゴリズムについて述べる．

第 6 章では，本研究の実験で扱う例題について説明する．はじめに，GP のベンチマーク問題として一般に扱われる関数同定問題（symbolic regression problem）に関して，使用する命令セットと対象とする関数の例題を示す．次に，実応用を想定したアセンブリプログラムの進化を扱う問題として，PIC マイコンで使用可能な命令セットについて説明し，PIC アセンブリプログラムを用いて実行する数値計算プログラムと論理演算プログラムの例題を示す．最後に，宇宙機への適用を念頭においた問題として宇宙放射線によるビット反転が起こる問題を説明し，その問題を想定した実験環境を説明する．

第 7 章では，TAEA を用いる実験とその結果について述べる．一つ目の実験では，アセンブリプログラムの進化を扱う例題において，従来手法と TDTS を用いる TAEA と TDTS を用いない TAEA を比較し，TDTS を用いる TAEA の有効性を示すとともに，これらを適応型 TAEA（TAEA$^+$）と比較し，TAEA$^+$ の有効性を検証する．従来手法としては同期 GP の steady-state GP（SSGP）と $(\mu + \lambda)$-GP，従来の非同期 GP である ASSGP を扱う．二つ目の実験では，ビット反転が生じる環境下でのアセンブリプログラムの進化を扱う例題に対して，ビット反転率を複数設定し，ビット反転が起こる環境下で TDTS を用いる TAEA が目的を達成可能なプログラムを進化可能であるかを検証する．最後に，三つ目の実験では，GP の一般的なベンチマーク問題である関数同定問題において，従来手法である同期 GP の $(\mu+\lambda)$-GP，従来の非同期 GP である非同期 steady-state GP（ASSGP），TAEA$^+$ を比較し，TAEA$^+$ の有効性を示す．特に，ここでは解の評価時間にばらつきがある環境として，(1) 解ごとに計算速度が異なる場合，(2) 評価が完了しない個体が含まれる場合を想定する．

第 8 章では，ARE-EA を用いる実験とその結果について述べる．一つ目の実験では，アセンブリプログラムの進化を扱う例題において従来手法である同期 GP の steady-state GP (SSGP)，TAEA，ARE-EA を比較し，ARE-EA の有効性を示す．二つ目の実験では，ビット反転が生じる環境下でのアセンブリプログラムの進化を扱う例題に対して，ビット

反転率を複数設定し，ビット反転が起こる環境下で ARE-EA が目的を達成可能なプログラムを進化可能であるかを検証する．最後に，三つ目の実験では，GP の一般的なベンチマーク問題である関数同定問題において，従来手法である同期 GP の $(\mu + \lambda)$-GP，従来の非同期 GP である非同期 steady-state GP（ASSGP），ARE-EA を比較し，ARE-EA の有効性を示す．特に，ここでは解の評価時間にばらつきがある環境として，(1) 解ごとに計算速度が異なる場合，(2) 評価が完了しない個体が含まれる場合を想定する．続いて，ARE-EA に含まれるパラメータの影響を分析するためにパラメータを変化させた場合の性能の違いを明らかにする．

最後に，第 9 章で本論文のまとめを述べ，今後の課題と展望を示す．

第 2 章

進化的アルゴリズム

進化的アルゴリズム（Evolutionary Algorithm: EA）は生物の進化のメカニズムをもとにしたメタヒューリスティックな最適化手法の一手法である．EA は，評価関数の設計のみに基づいて解探索が可能なため，解空間のモデル化や前提なく適用可能であるという特徴がある．

本章では，EA の代表的な手法である遺伝的アルゴリズム，遺伝的プログラミングについてそれぞれ説明し，その後，並列計算環境を用いる EA について説明する．

2.1　同期進化的アルゴリズム

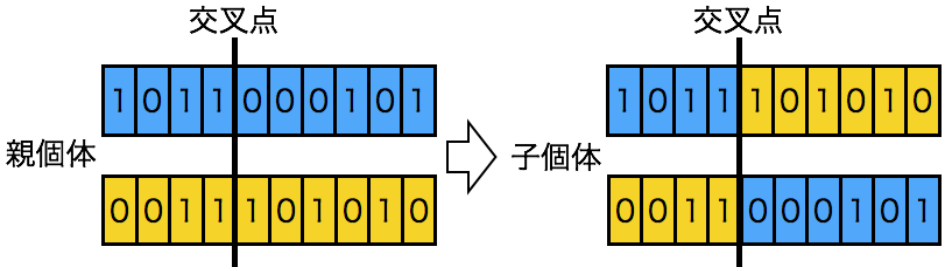
2.1.1　遺伝的アルゴリズム

遺伝的アルゴリズム（Genetic Algorithm: GA）[2] は，代表的な EA の一手法である．GA において，解は遺伝子の形で表され，問題によってビット列，実数値列，整数値列，またはそれらを組み合わせて表現される．例えば，組合せ最適化問題では組み合わせを決定する要素分の長さを持つビット列で遺伝子を表し，選択する要素を 1，選択しない要素を 0 として表現し，関数の最大値を求める問題では，探索空間上の座標を実数値列として遺伝子を表す．また，遺伝子の各要素は遺伝子座と呼ばれる．

Algorithm 1 に，GA の基本的な流れを示す．まずはじめに，GA ではあらかじめ設定された母集団サイズ分の解をランダムに生成し，初期母集団とする（1 行目）．次に，母集団中のすべての解の適応度を評価関数に基づいて算出する（2 行目）．そして，算出された適応度をもとに次世代の解を生成するための親個体を選択する（3 行目）．親個体の選択は適応度の高い個体ほどより選択確率が高くなるように設計される．具体的には，全個体の適応度の比率から選択確率を決定するルーレット選択，母集団中の適応度の順位から選択確率を決定するランキング選択，母集団中からランダムに選択した 2 個体のうち適合度の高い個体を選択するトーナメント選択などがある．選択された親個体から交叉や突然変

Algorithm 1 GA の基本的な流れ

1: 初期母集団の生成
2: 母集団中の解を評価
3: 適応度に基づく親個体選択
4: 親個体から遺伝的操作（交叉，突然変異等）によって子個体を生成
5: 次世代母集団を生成
6: 終了条件を満たさなければ 2. に戻る



(a) 交叉

(b) 突然変異

図 2.1　遺伝的操作

異と呼ばれる遺伝的操作を通して新しい子個体を生成する（4 行目）．交叉は 2 体の親個体を組み合わせて子個体を生成する方法であり，突然変異は親個体の一部をランダムに変更した子個体を生成する方法である．例えば，遺伝子がビット列で表現される場合，交叉は図 2.1(a) に示すように遺伝子のランダムな点（交叉点）を境に 2 つの親個体を組み替えて新しい子個体を生成し，突然変異は図 2.1(b) に示すようにランダムな遺伝子座を変更する（図では 0 を 1 に反転）．これらの遺伝的操作はあらかじめ定められた確率（パラメータ）に基づいて実行され，交叉を実行する確率を交叉率，突然変異を実行する確率を突然変異率と呼ぶ．最後に，遺伝的操作によって生成された子個体と現在の母集団から次世代の母集団を生成する（5 行目）．次世代母集団の生成方法としては，現在の母集団の上位数個体以外を生成された子個体と入れ替えるエリート選択法や母集団と子個体を合わせた中から上位母集団サイズ分の個体を選択する $(\mu + \lambda)$-GA，子個体を 1 個体ずつ生成し親個体と入れ替える steady-state GA（SSGA）などがある．以降，これらの操作をあらかじめ定めた終了条件（世代数や最良解の適応度）を満たすまで繰り返す（6 行目）ことで最適解を探索する．

以下で具体的に，$(\mu + \lambda)$-GA と SSGA について説明する．まず，$(\mu + \lambda)$-GA の概略

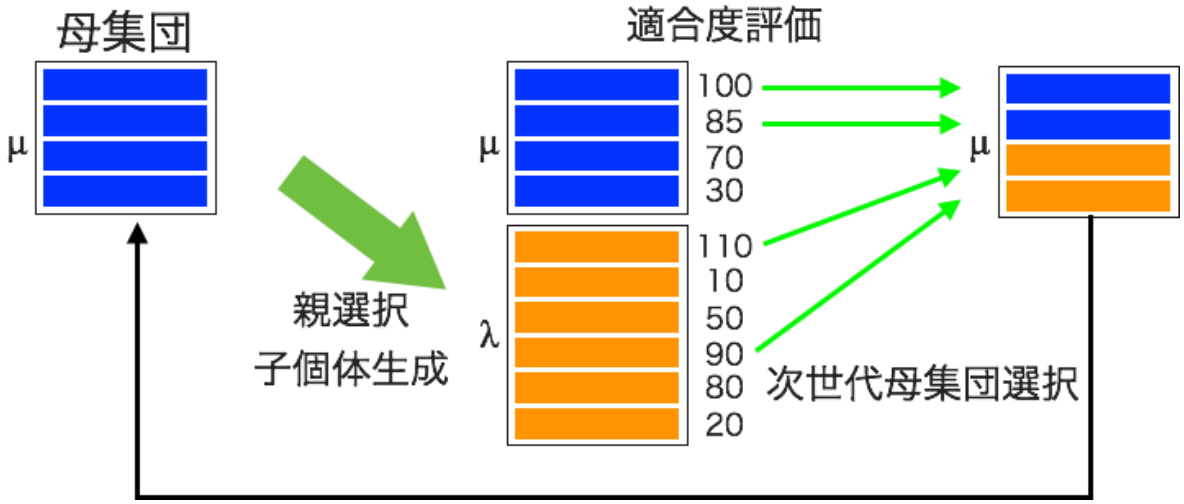


図 2.2　$(\mu + \lambda)$-GA の概略図

Algorithm 2 $(\mu + \lambda)$-GA のアルゴリズム

1: μ 個の初期母集団を生成
2: λ 個の親個体を選択
3: 選択した親個体から遺伝的操作を通して λ 個の子個体を生成
4: 母集団と子個体を合わせた $(\mu + \lambda)$ 個体から μ 個の親個体を次世代母集団として選択
5: 終了条件を満たさなければ 2. に戻る

Algorithm 3 SSGA のアルゴリズム

1: 初期母集団を生成
2: トーナメント選択を 2 回行い，勝者を *winner*，敗者を *loser* とし，*winner* の 2 個体を親個体とする
3: 選択した親個体から遺伝的操作を通して 2 個の子個体を生成
4: *loser* を母集団から削除し，2 個の子個体を母集団に加える
5: 終了条件を満たさなければ 2. に戻る

図を図 2.2，その流れを Algorithm 2 に示す．図 2.2 において，各長方形は個体を表す．$(\mu + \lambda)$-GA では，母集団サイズを μ，生成子個体数を λ とし，はじめに μ 個体の母集団の中から λ 個の親個体を選択する．その後，λ 個の親個体から遺伝的操作によって子個体を λ 個生成する．最後に μ 個の親個体と λ 個の子個体を合わせた $(\mu + \lambda)$ 個の個体の中から適合度の上位 μ 個を次世代の母集団とする．以降，この操作を終了条件を満たすまで繰り返す．

次に，SSGA について説明する．SSGA の概略図を図 2.3，その流れを Algorithm 3 に示す． SSGA では，親集団からランダムに 2 個体を選択し，適合度の優れた個体を選択

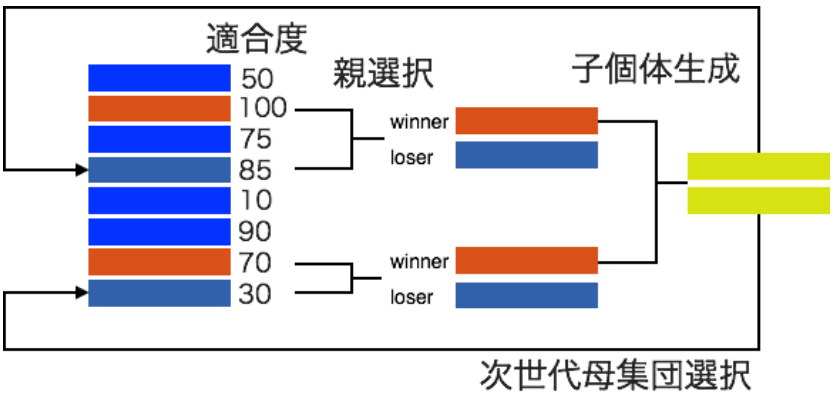


図 2.3　SSGA の概略図

するトーナメント選択を 2 回実行し，親個体を 2 個体選択する．この時，トーナメント選択の勝者（親個体）を *winner*，敗者を *loser* とする．選択された 2 個の親個体から，遺伝的操作によって子個体を 2 個体生成する．次に，2 個体の *loser* を母集団から削除し，代わりに生成された子個体を母集団に追加する．以降，この操作を終了条件を満たすまで繰り返す．

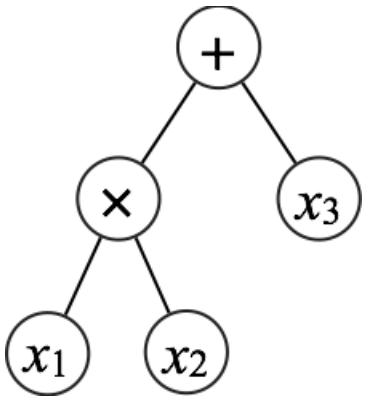
2.1.2　遺伝的プログラミング

遺伝的プログラミング（Genetic Programming: GP）[3] は EA の中でも特に関数やプログラムを最適化する代表的な一手法である．基本的なアルゴリズムは GA と同様であるが，GA における遺伝子座にプログラムの一つの命令が対応し，遺伝子が一連の処理を実行するプログラムとして表される．GP はプログラムの表現法によっていくつかの種類に分類される．具体的には，命令とその引数を木構造で表現した最も一般的な Tree GP（TGP）（図 2.4(a)），命令語の列として配列構造で表現した Linear GP（LGP）[23, 24, 25]（図 2.4(b)），有向グラフを用いて表現した Cartesian GP（CGP）[26]（図 2.4(c)）などが代表的である．

本節では，GP の中でも最も一般的な TGP，および本研究で扱う LGP をもとに GP の概略を説明する．

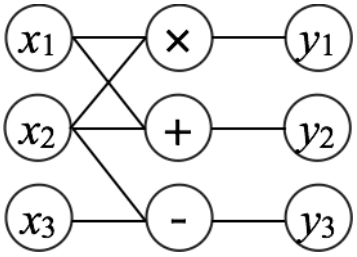
Tree GP （TGP）

Tree GP（TGP）は J. Koza によって提案された最も代表的な GP であり，プログラムの表現として木構造を用いる．プログラムを表現する木構造の各記号はノードと呼ばれ，木の根にあたるノードをルートノード，木の末端（葉）にあたるノードを終端ノード（または終端記号），それ以外を非終端ノード（または非終端ノード）と呼ぶ．例えば，図 2.4(a) では，木の根にあたる「+」記号がルートノード，木の末端にあたる「x_1」，「x_2」，「x_3」



(a) 木構造によるプログラム表現（TGP）

(b) 配列構造によるプログラム表現（LGP）



(c) グラフ構造によるプログラム表現（CGP）

図 2.4　GP におけるプログラム表現

Algorithm 4 TGP の基本的な流れ

1: 初期母集団の生成
2: 母集団中の解を評価
3: 適応度に基づく親個体選択
4: 親個体から遺伝的操作（交叉，突然変異，逆位）によって子個体を生成
5: 次世代母集団を生成
6: 終了条件を満たさなければ 2. に戻る

記号が終端ノード，「x_1」と「x_2」をつなぐ「×」記号が非終端ノードとなる．また，非終端ノードと終端ノードからなる木の一部を部分木と呼ぶ．例えば，図 2.4(a) では，「×」，「x_1」，「x_2」からなる木の一部，または「x_3」などが部分木となる．プログラムは深さ優先探索で各ノードに示された命令を実行する．例えば，図 2.4(a) では，「$(x_1 \times x_2) + x_3$」が実行される．

TGP の流れを Algorithm 4 に示す．基本的な流れは GA と同様であり，母集団の初期化，評価，親個体選択，子個体生成と世代交代を繰り返す．TGP では，遺伝子として木構造を用いるため，それに合わせた遺伝的操作が実行される．TGP における交叉では，

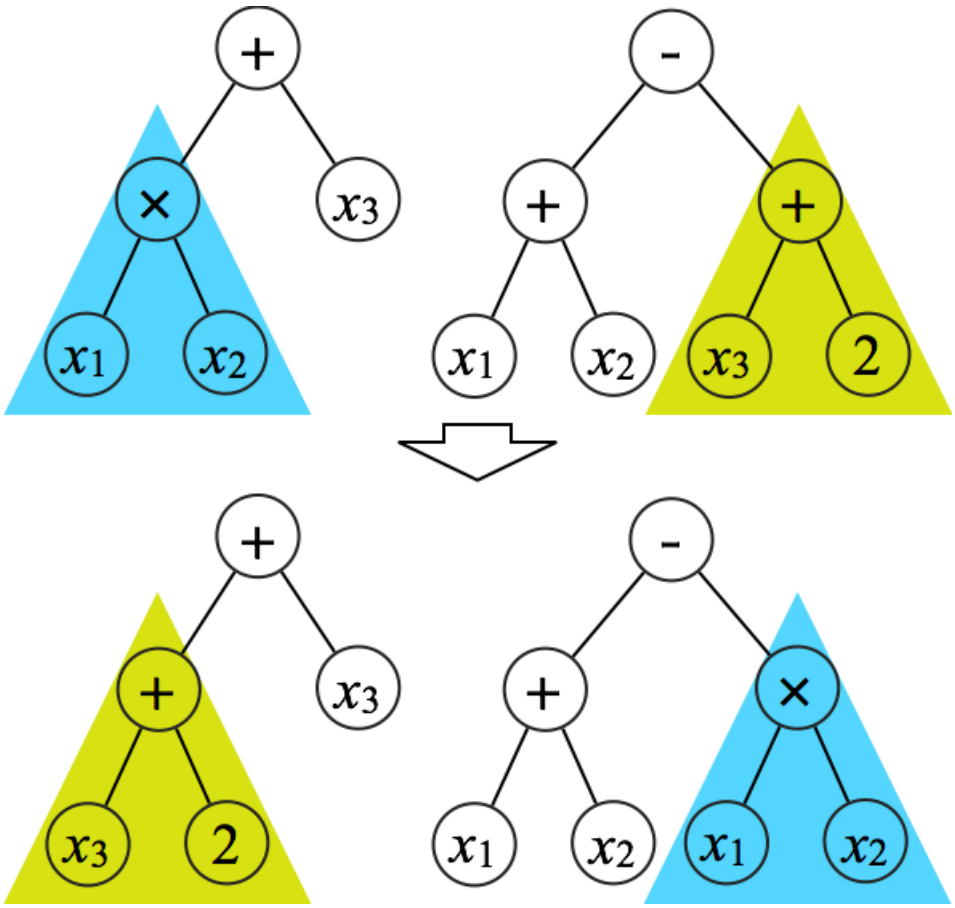
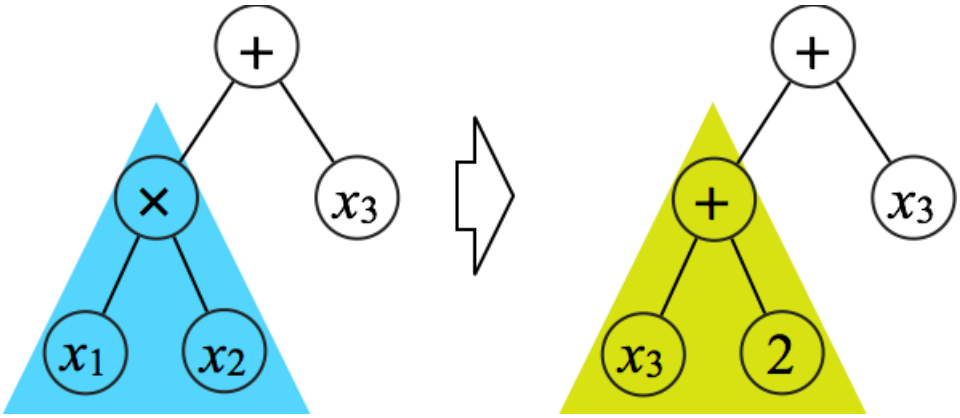
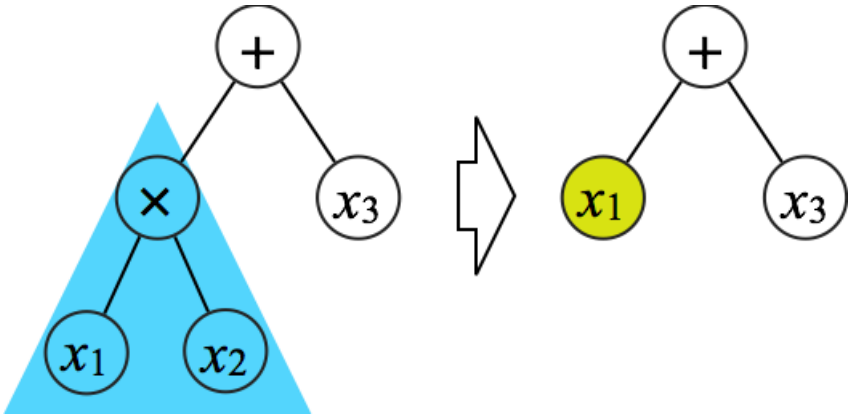


図 2.5　TGP における交叉

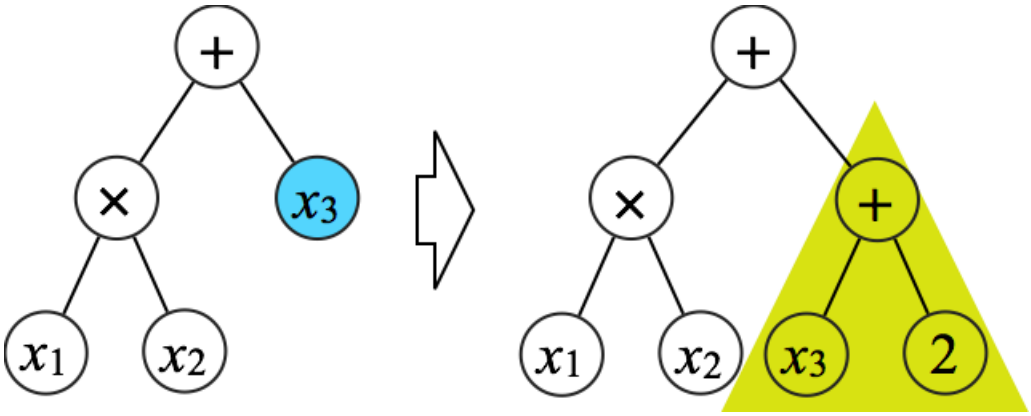
図 2.5 に示すように，親個体 2 個体の部分木の交換によって新しい子個体が生成される．この時，交叉の対象となる部分木のルートノード（図 2.5 における「×」と「+」）を交叉点と呼ぶ．突然変異は，木構造にランダムな変更を加えるが，図 2.6 に示すように，(1) 部分木から部分木への突然変異，(2) 部分木から終端ノードへの突然変異，(3) 終端記号から部分木への突然変異，(4) 終端記号から終端記号への突然変異，(5) 非終端記号から非終端記号への突然変異に分類される．(1) 部分木から部分木への突然変異は，図 2.6(a) に示すようにランダムに選択した部分木をランダムに生成した部分木で置き換える．(2) 部分木から終端ノードへの突然変異は，図 2.6(b) に示すようにランダムに選択した部分木をランダムな終端ノードで置き換える．(3) 終端記号から部分木への突然変異は，図 2.6(c) に示すようにランダムに選択した終端ノードをランダムに生成した部分木で置き換える．(4) 終端記号から終端記号への突然変異は，図 2.6(d) に示すようにランダムに選択した終端ノードをランダムな終端ノードで置き換える．(5) 非終端記号から非終端記号への突然変異は，図 2.6(e) に示すようにランダムに選択した非終端ノードを引数の個数が同一のランダムな非終端ノードで置き換える． GA にはない TGP の特徴的な遺伝的操作として，逆位と呼ばれる操作がある．逆位では，図 2.7 に示すように，一つの木に含まれる部



(a) 部分木から部分木への突然変異

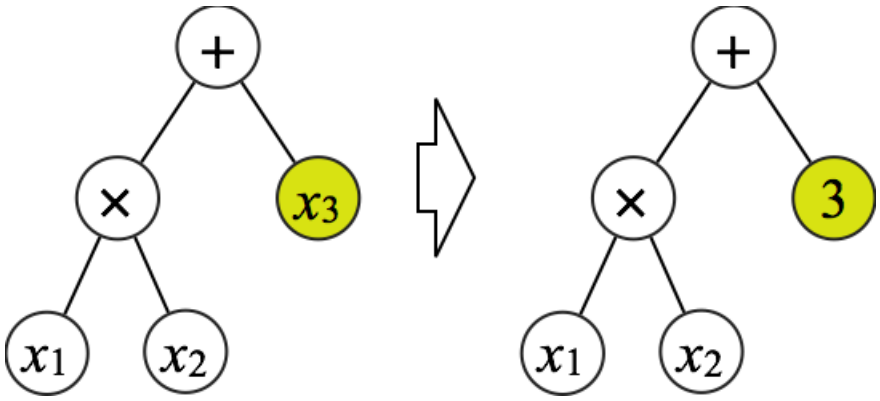


(b) 部分木から終端ノードへの突然変異

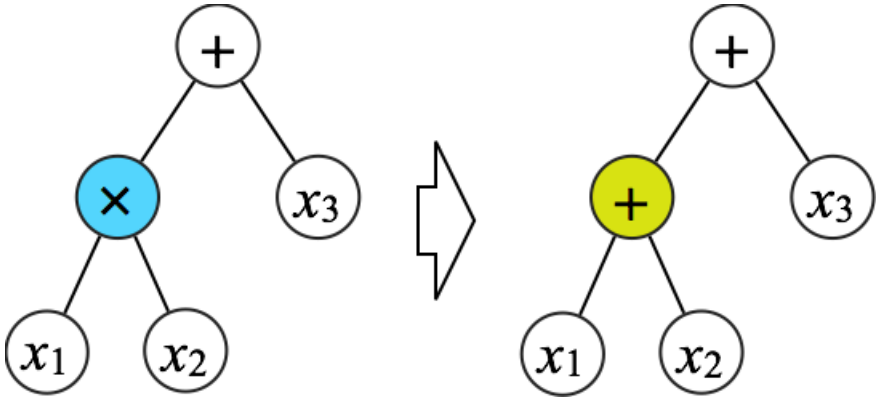


(c) 終端ノードから部分木への突然変異

図 2.6　TGP における突然変異（1/2）



(d) 終端ノードから終端ノードへの突然変異



(e) 非終端ノードから非終端ノードへの突然変異

図 2.6　TGP における突然変異（2/2）

分木同士の位置を交換する．

Linear GP （LGP）

Linear GP（LGP）は，TGP においてプログラムを評価（実行）する際に木構造を解釈（interpret）する必要があるという問題に対し，計算機で直接実行可能な機械語列，あるいはそれに準ずる命令語列のまま進化させることによって高速化と実行可能なコードの生成を目指した GP である [27, 28, 24]．LGP の流れを Algorithm 5 に示す．基本的な流れは GA と同様であり，母集団の初期化，評価，親個体選択，子個体生成と世代交代を繰り返す．LGP では，遺伝子が命令語の列として表現され，各命令語は図 2.8 に示すように実行する操作の内容を表すオペコード（opcode）と実行対象となるレジスタ（変数）を指すオペランド（operand）からなり，一般に「$opcode\ \ dst\ src1\ src2$」のような形で表される．ここで，$opcode$ はオペコード，dst は実行結果を出力するレジスタ，$src1$，$src2$ はそれぞれオペコードの入力となるレジスタを表す．例えば，図 2.8 に示す「$ADD\ \ r0\ r1\ r2$」は「$r1$ と $r2$ の値を加算して $r0$ に代入する（$r0 \leftarrow r1 + r2$）を表す．GA と LGP では，

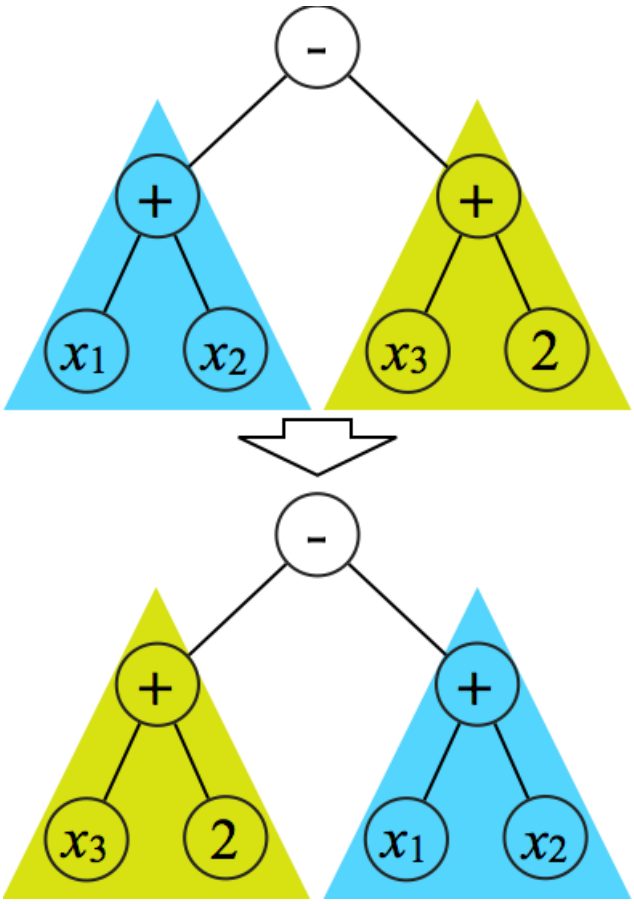


図 2.7　TGP における逆位

Algorithm 5 LGP の基本的な流れ

1: 初期母集団の生成
2: 母集団中の解を評価
3: 適応度に基づく親個体選択
4: 親個体から遺伝的操作（交叉，突然変異，命令語の挿入，削除等）によって子個体を生成
5: 次世代母集団を生成
6: 終了条件を満たさなければ 2. に戻る

GA の遺伝子長は固定長であるのに対し，LGP では遺伝子がプログラムを表すため可変長であるため，多様な遺伝子長を持つ初期集団の生成，および遺伝子長を変化させる遺伝的操作が行われるという点で大きく異る．初期集団の生成において，GA では遺伝子長は均一であるため，遺伝子の内部の値のみを初期化すれば良いが，LGP の場合には可変長の遺伝子を扱うため，遺伝子長についてもランダムに生成する必要がある．LGP では，限られた範囲の比較的短いサイズの遺伝子を初期集団としてランダムに生成して初期集団と

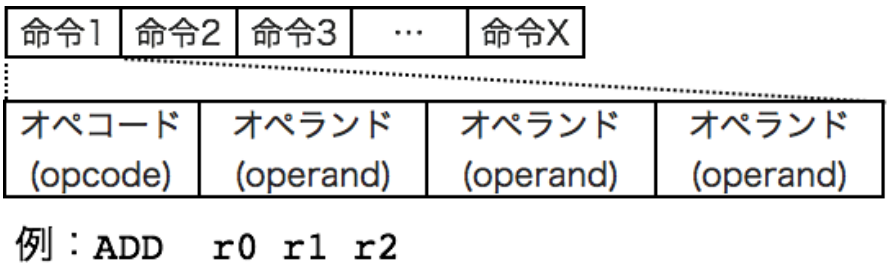


図 2.8　LGP における遺伝子と命令語

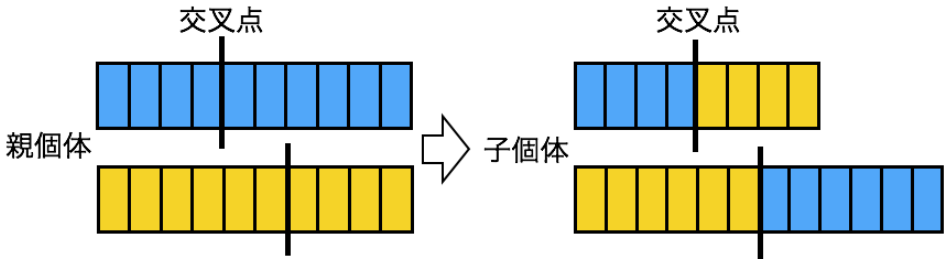
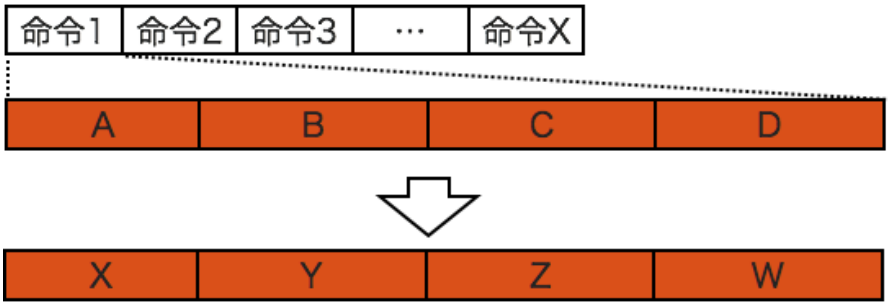


図 2.9　LGP における交叉

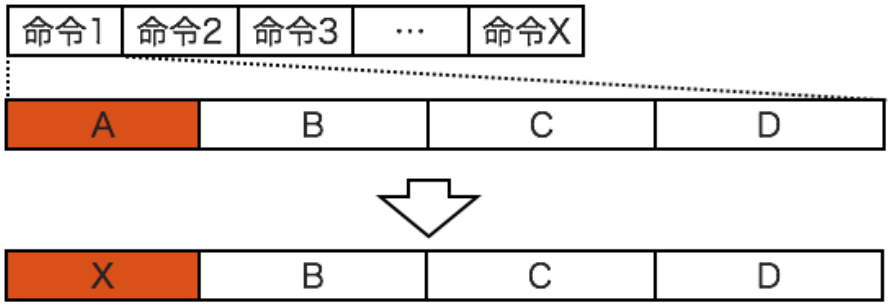
するのが一般的である．次に交叉について，GA では 2 つの親個体で交叉点は同一であったが，LGP では図 2.9 に示すように異なる交叉点を設定することができる．これにより，親個体とは異なる遺伝子長を持つ子個体が生成される．突然変異については，GA と同様にプログラム中のある命令語に対してランダムな変更を加えるが，その変更方法によって 2 種類に分類される．一つ目は，図 2.10(a) に示すようにオペコードとオペランドを含む命令語全体をランダムに変更する方法，もう一つは図 2.10(b) に示すようにオペランド，もしくは一部のオペコードのみを変化させる方法である．また，LGP では交叉，突然変異に加えて命令語の追加，削除という操作が実行される．具体的には，図 2.11 に示すように，プログラムのランダムな位置にランダムな命令を追加，あるいはランダムな命令を削除することによって子個体が生成される．

2.2　並列型進化的アルゴリズム

GA や GP を含む EA では，母集団中の複数の解の評価値を算出する必要がある．近年の計算機技術の発展に伴い，多数の計算機を用いて高速に計算をする並列化技術が着目されており，進化計算の分野においても多数の個体（解）を並列に評価し，効率的な解探索を実現する並列型 EA（Parallel EA: PEA）が提案されている [12, 13, 14, 15]．並列型 EA は，その並列化の方法によって，(1)Master-Slave 型，(2) 島モデル型，(3) セルラー型の 3 種類に分類される．以下，これらの並列型 EA について説明する．

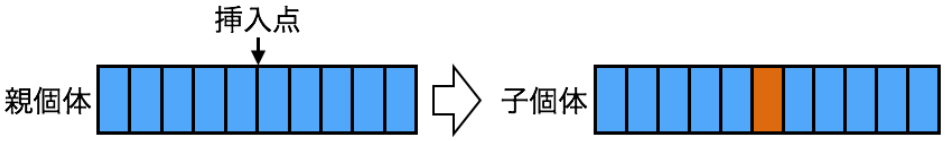


(a) 命令語全体の突然変異

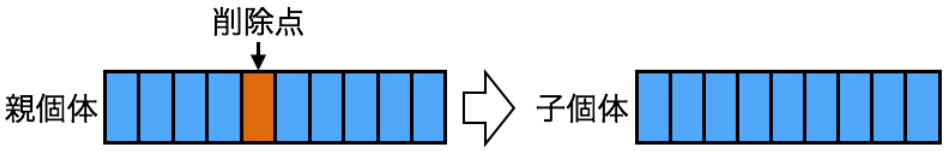


(b) 命令語の一部の突然変異

図 2.10　LGP における突然変異



(a) 命令の挿入



(b) 命令の削除

図 2.11　LGP における命令の挿入，削除

2.2.1　Master-Slave 型

Master-Slave 型の PEA は，最も単純な PEA であり，図 2.12 に示すように，1 台の Master ノードと複数台の Slave ノードからなり，Slave ノードがそれぞれ解の評価を実行し，Master ノードが EA における評価以外の親選択，子個体生成，世代交代などを実行する．これにより，Slave ノードの台数分の解評価を同時に実行することができるため，最大 Slave ノードの台数倍の解探索速度を実現することが可能になる．Master-Slave 型

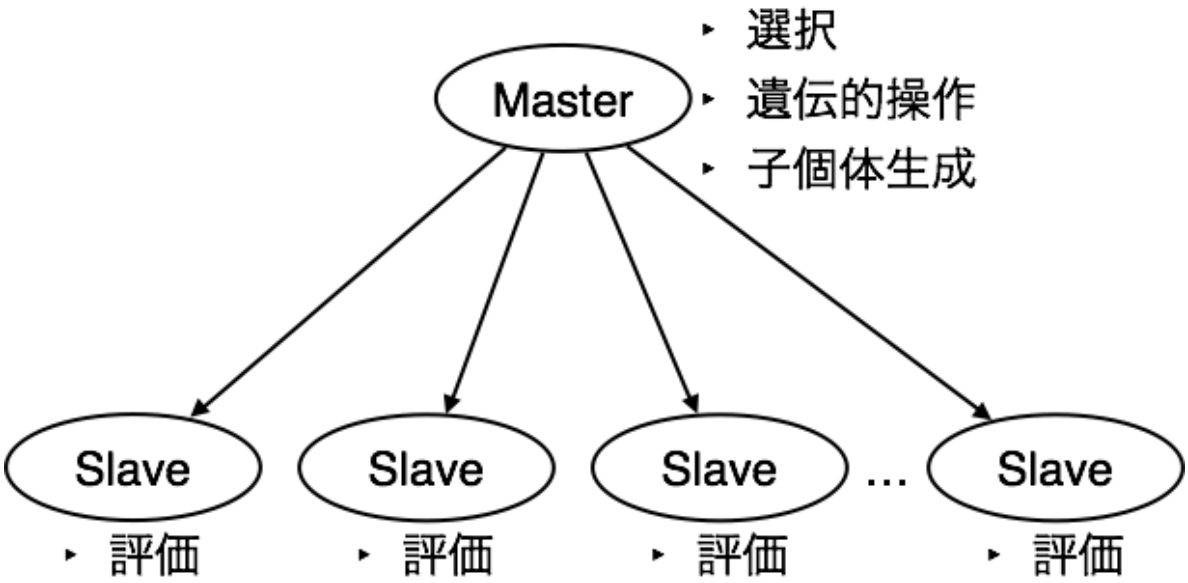


図 2.12　Master-Slave 型 PEA の概略図

Algorithm 6 島モデル型 PGA のアルゴリズム

1: 島ごとに初期母集団を生成
2: 島ごとに進化（選択，遺伝的操作，世代交代）
3: 一定世代経過後，島の間で優良個体を移住
4: 終了条件を満たさなければ 2. に戻る

PEA は，解の評価を並列化する以外は，一般的な EA と同様の動作となる．

2.2.2　島モデル型

島モデル（Island-model）型の PEA[29] は，図 2.13 に示すように母集団を複数の島（サブ母集団：subpopulation）に分割して，それぞれの島で最適化を行う方法である．各島の間では，一定間隔ごとに優良個体を他の島に移動させる移住（migration）という操作が行われ，島同士での解の情報交換がなされる．また，島モデル型 PEA では，島ごとに遺伝的パラメータや母集団サイズを変更可能である．具体的な島モデル型 PEA の流れを Algorithm 6 に示す．島モデル型 PEA では，母集団を複数の島での最適化と移住を繰り返すことにより，解の多様性を維持することが可能である．

2.2.3　セルラー型

セルラー型（cellular，または fine-grained）の PEA[30, 31] は，各解は平面上のセルにそれぞれ配置され，近隣のセルとの間での情報交換によって選択や遺伝的操作を実行す

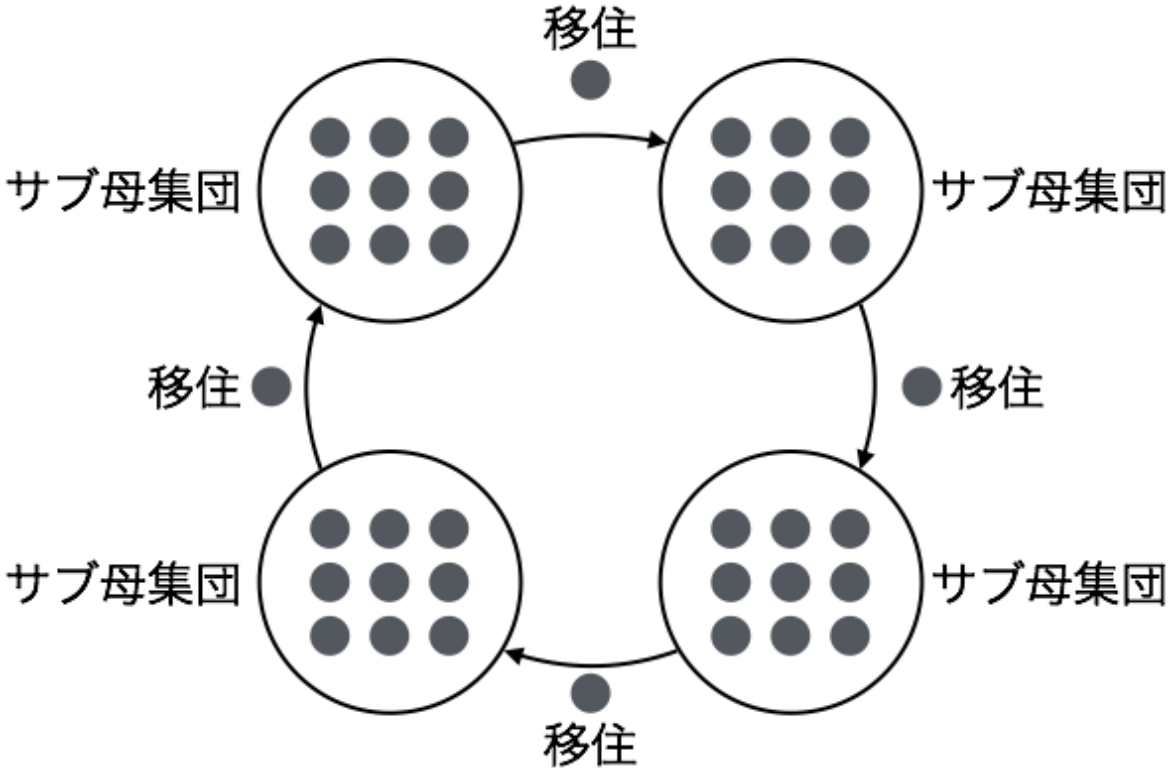


図 2.13　島モデル型 PEA の概略図

る．具体的には，図 2.14 に示すように，各解が平面上の各セル（図 2.14 上の丸）に 1 個体ずつ割り当てられ，その位置関係に基づいて周囲の個体との間で親選択や交叉相手が決定される．例えば，図 2.14 で近隣の 4 個体との間で進化する場合，2 行 3 列目の個体は上下左右 4 個体と自身を合わせた 5 個体の中から親個体を選択し，遺伝的操作によって子個体を生成する．セルラー型 PEA では島モデル型 PEA における移住のように解の平面上での配置を変更するような操作は行われず，優良解の情報は隣接する解の進化を通じて間接的に伝搬する．

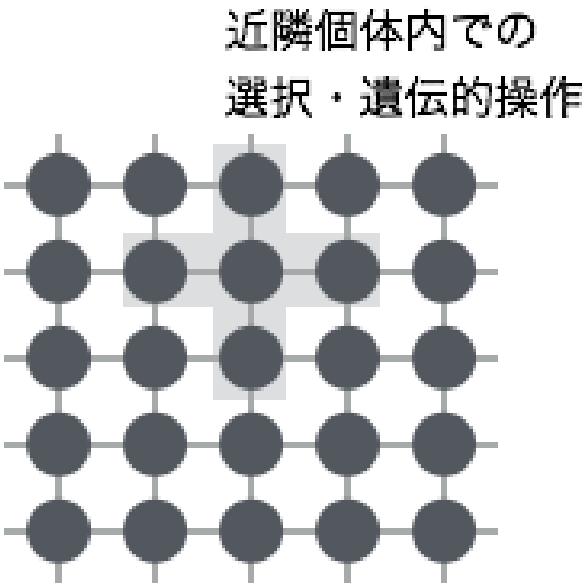


図 2.14　セルラー型 PEA の概略図

第 3 章

非同期進化的アルゴリズムの関連研究

3.1　従来の進化的アルゴリズムの問題点

従来の一般的な EA では，世代（*generation*）に基づいて解を進化させるため，母集団内の全解の評価が同時に得られる必要がある．例えば，$(\mu + \lambda)$-GA では，生成された λ 個の解の評価値を算出してからでなければ次世代の μ 個の母集団を選択することができない．これに対し，PEA では解または母集団を並列に評価することで解評価にかかる時間を削減している．しかし，PEA において解を並列に評価した場合でも，評価時間にばらつきがある場合には最も評価時間の遅い個体の評価が完了してからでないと次世代の母集団を生成できず，計算時間の無駄が大きくなるという問題がある．例えば，図 3.1 に示すように，各解が図中の矢印で示す長さの評価時間が必要である場合，個体 5 を除く個体は比較的短い時間で評価が完了するのに対し，個体 5 は評価に多くの時間が必要になる．この場合，次世代母集団を決定するためには最も評価時間の長い個体 5 を待つ必要がある．EA で扱う最適化問題において解の評価時間が均一でない状況は容易に起こりえる．例えば，(1) 解の性質によって評価時間が異なる場合や (2) 並列計算環境において計算機の性能が異なる場合などである．(1) 解の性質によって評価時間が異なる場合は，具体的には図 3.2 に示すように，GP でプログラムを進化させる際のプログラム長の差異やループ回数の差異による実行ステップ数の差異，ロボットの歩行制御最適化の場合の歩行距離による評価時間差などが考えられる．図 3.3 に示すような (2) 並列計算環境において計算機の性能が異なる場合では，同一の解であっても評価に用いる計算機によって評価時間差が生じる．

また，扱う問題によっては評価時間が長くなるだけではなく，図 3.4 に示すように個体の評価自体が完了しないという状況も考えられる．例えば，並列計算環境において計算機

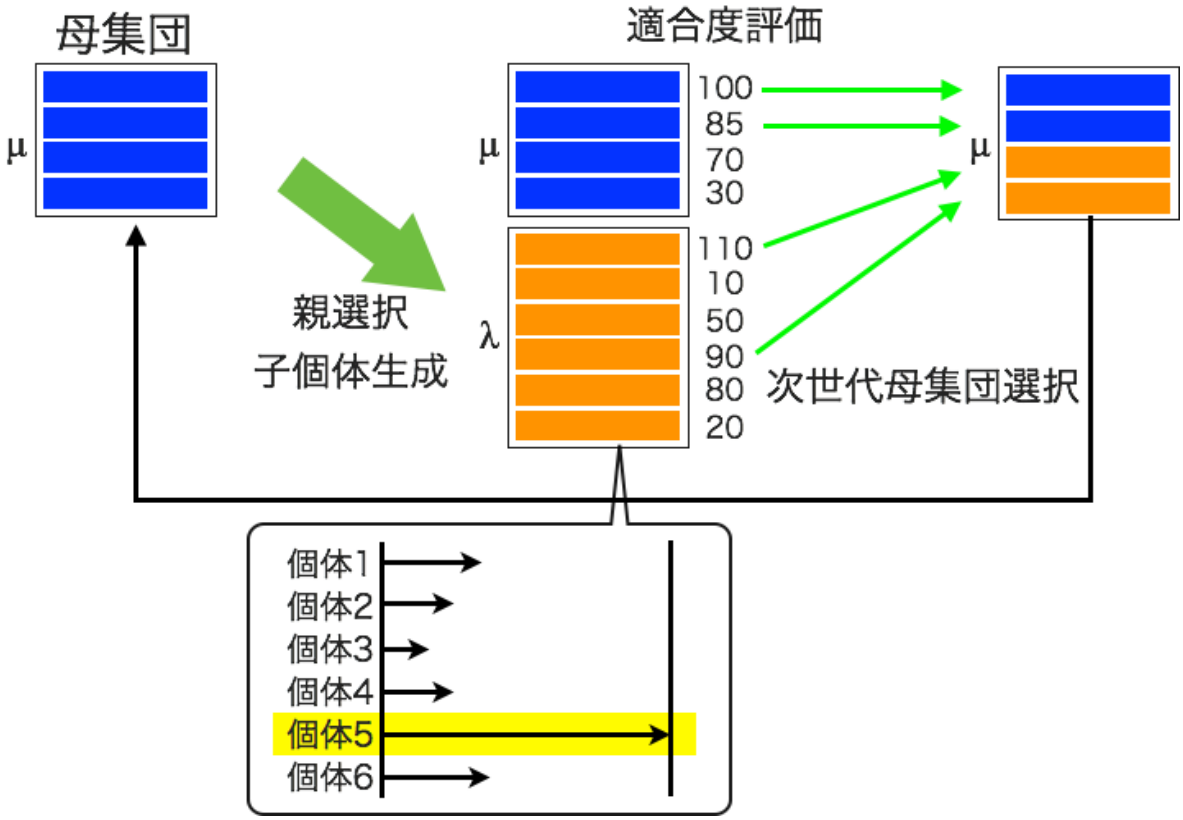


図 3.1　評価時間にばらつきがある場合の EA の問題点

```
while(i<10){
   ++i;
}
```

```
while(i<100){
   ++i;
}
```

(a) 実行ステップ数の差異

(b) ロボットの歩行制御最適化

図 3.2　解の性質によって評価時間が異なる例

間の通信エラーが発生する場合や，GP において個体として進化させるコンピュータプログラムに無限ループが含まれる場合などである．個体の評価が得られない場合，従来の EA においては，評価の待機時間に上限を設け，その時間を経過しても個体が得られない場合には評価不能として打ち切り，適合度に最低値を与える方法が一般的である．しかし，解の性質が未知の問題においては適切な待機時間を設定することが困難であり，待機

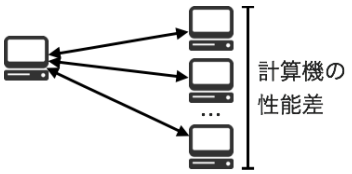
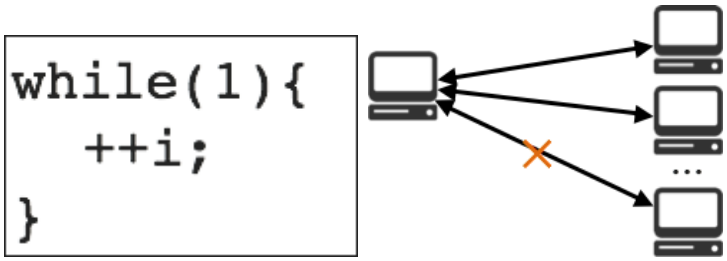


図 3.3　並列計算機の性能差



(a) プログラムの無限ループ　　(b) 並列計算機の通信エラー

図 3.4　評価が完了しない例

時間の設定が解探索性能に大きな影響を与える．待機時間が長すぎる場合には計算時間が無駄になり，逆に短すぎる場合には評価が完了するはずの個体を無視することになる．そのため，従来の EA ではこれらの状況への対処が困難である．

3.2　非同期進化的アルゴリズム

上記の問題に対して有効な方法として，近年では集団ではなく個体ごとに独立に進化させる方法 [6, 5, 8] や全個体の評価値を待たずに非同期（*asynchrnous*）に解を進化させる非同期 EA が提案されている [17, 18, 19, 20, 21, 16]．

3.2.1　差分進化

差分進化（Differential Evolution：DE）[6] は，実数値の最適化問題を扱う EA であり，単純なベクトル演算による更新によって解を探索する．DE では，Algorithm 7 に示す手順で解を更新する．式 (3.1) において，$\mathbf{v}_i$ は突然変異ベクトル，$\mathbf{x}_{r1}$，$\mathbf{x}_{r2}$，$\mathbf{x}_{r3}$ はそれぞれ異なる個体（$i \neq r1 \neq r2 \neq r3$）を表し，$F$ はパラメータ（$0 \leq F \leq 2$）である．式 (3.2) において，$\mathbf{u}_i$ は生成される子個体，j は遺伝子中の j 番目の要素，$rand()$ は 0 から 1 の乱数，I_{rand} は遺伝子中のランダムに選択された要素をそれぞれ表し，CR は交叉率を表すパラメータ（$0 \leq CR \leq 1$）である．具体的に，DE における子個体生成の概略を図 3.5 に示す．図 3.5 のように，生成される子個体 u_i は親個体 x_i と突然変異ベクトル v_i の各要素を組み合わせた値をとる．突然変異ベクトルの生成の際に $\mathbf{x}_{r2}$，$\mathbf{x}_{r3}$ は基本的にランダムに選択されるが，$\mathbf{x}_{r1}$ はランダムに選択する場合と集団中の最良個体を選択す

Algorithm 7 DE のアルゴリズム

1: 全個体を初期化
2: 全個体の評価値を算出
3: **for** i=1 から全個体分 **do**
4: 突然変異ベクトルを生成

$$\mathbf{v}_i = \mathbf{x}_{r1} + F \cdot (\mathbf{x}_{r2} - \mathbf{x}_{r3}) \tag{3.1}$$

5: 交叉により子個体を生成

$$\mathbf{u}_{i,j} = \begin{cases} \mathbf{v}_{i,j} & if\ rand() < CR\ OR\ j = I_{rand} \\ \mathbf{x}_{i,j} & otherwise \end{cases} \tag{3.2}$$

6: 子個体の評価値を算出
7: 親個体より子個体が優れていれば入れ替え
8: **end for**
9: 終了条件を満たさなければ 3. に戻る

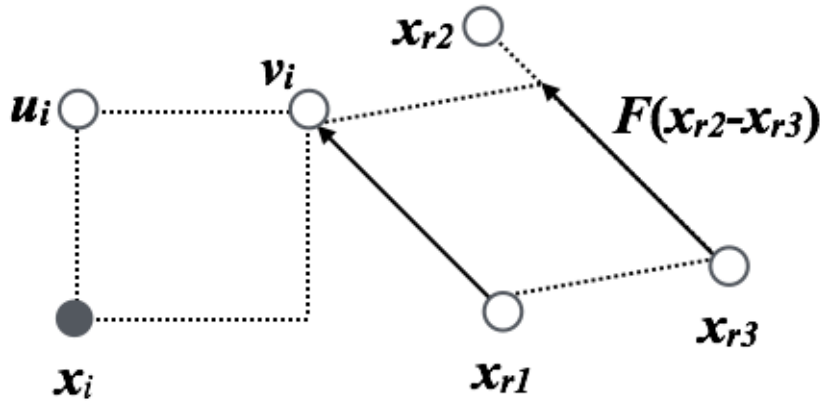


図 3.5　DE の概略図

る場合がある． DE は，Algorithm 7 中 6 行目に示すように親個体と子個体の 1 対 1 の比較のみで生存選択を行う．そのため，一般的には世代の概念を用いて全個体同期をとって進化をするが，それぞれ非同期に進化させることも可能である．具体的には，[13] で提案されているように，各個体を並列に評価し，評価が完了するごとにただし，突然変異ベクトルの生成の際に $\mathbf{x}_{r1}$ として集団内の最良個体を使用する方法では，全個体の評価値を算出してから最良個体を選択する必要があるため，同期的に進化させる必要がある．

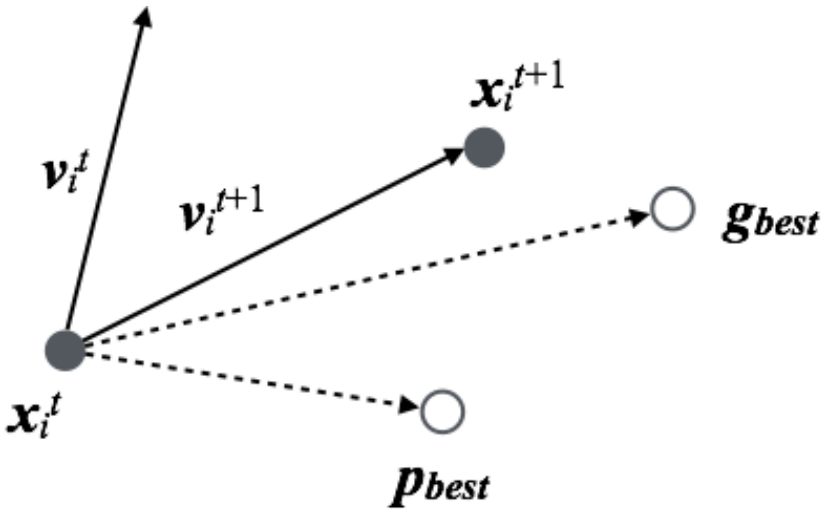


図 3.6　PSO における更新式の概略図

3.2.2　粒子群最適化

EA の中でも特に実数値の最適化として有名な手法として粒子群最適化（Particle Swarm Optimization：PSO）[5] がある．PSO の具体的な流れを Algorithm 8 に示す．PSO では解を粒子と呼び，各粒子 i は位置 $\mathbf{x}_i$ と速度 $\mathbf{v}_i$ を持つ．そして，各粒子は下記の更新式に従って世代ごとに位置と速度を更新する．

$$\mathbf{v}_i^{t+1} \leftarrow \mathbf{v}_i^t + w_1 \times rand() \times (\mathbf{p}_{best}^t - \mathbf{x}_i^t) + w_2 \times rand() \times (\mathbf{g}_{best}^t - \mathbf{x}_i^t) \quad (3.3)$$

$$\mathbf{x}_i^{t+1} \leftarrow \mathbf{x}_i^t + \mathbf{v}_i^{t+1} \quad (3.4)$$

ここで，式 (3.3) において，$\mathbf{p}_{best}$ は粒子 i がこれまで探索した中で最も評価値の高い位置（パーソナルベスト，personal best），$\mathbf{g}_{best}$ は全粒子の中で最も評価値の高い位置（グローバルベスト，global best）を表し，w_1，w_2 は重みパラメータ，$rand()$ は 0 から 1 の乱数である．上記更新式を図式化すると図 3.6 のようになる．図 3.6 に示すように，各粒子の速度 $\mathbf{v}_i^{t+1}$ は現時点での速度 $\mathbf{v}_i^t$ と現在の位置 $\mathbf{x}_i^t$ からパーソナルベスト $\mathbf{p}_{best}$，グローバルベスト $\mathbf{g}_{best}$ へのベクトルの重み和で決定され，その速度に基づいて次時点での位置 $\mathbf{x}_i^{t+1}$ が決定する． PSO は，生存選択は行わず，必ず現在の位置 $\mathbf{x}_i^t$ と式 (3.3) によって決まる速度 $\mathbf{v}_i^{t+1}$ によって次世代の位置 $\mathbf{x}_i^{t+1}$ が決定するため，DE と同様にそれぞれの粒子は独立に更新可能である．しかし，全粒子の評価値をもとに探索中の最良解をグローバルベスト $\mathbf{g}_{best}$ を更新する必要があるため，全粒子の同期をとる必要がある．

Algorithm 8 PSO のアルゴリズム

1: 全粒子の位置 $\mathbf{x}_i$ と速度 $\mathbf{v}_i$ を初期化
2: $\mathbf{p}_{best}$ を現在位置で初期化
3: 全粒子の評価値を算出
4: 各粒子に対し，$\mathbf{p}_{best}$ より現在位置の評価が優れていれば更新
5: 全粒子の評価値の最大値が $\mathbf{g}_{best}$ より優れていれば更新
6: 式 3.3，3.4 に基づいて全粒子の位置と速度を更新
7: 終了条件を満たさなければ 3. に戻る

3.2.3 MOEA/D

MOEA/D（MOEA on decomposition）[8] は，多目的 EA の一手法であり図 3.7 に示すように多目的の探索空間を複数の集約関数ベクトルで分割し，集約関数ベクトルごとに解を最適化する．MOEA/D では，親選択を現在対象としている集約関数ベクトルの解とその周辺の近隣の集約関数ベクトル（近傍ベクトル）の中から選択する．また，生成された子個体は各集約関数ベクトルにおいて現在属している解と比較し，集約関数値が優れていればその集約関数に属する解として入れ替える．MOEA/D では，子個体を生成する際に近隣ベクトルに属する解のみを利用するため，そこに含まれる解の評価のみで子個体を生成することが可能であるが，子個体の生存選択の際にすべての集約関数の解と比較する必要があるため，生存選択のためにすべての解の評価が必要となる．

3.2.4 非同期粒子群最適化

Carlisle らは，PSO を非同期に拡張した非同期 PSO（asynchronous PSO：APSO）を提案している [18]．従来の PSO では，グローバルベスト $\mathbf{g}_{best}$ は全粒子の評価値をもとに決定するため，他の EA と同様に世代に基づいて解を更新する．そのため，解の評価時間差がある場合に最も評価時間の長い個体を待機する必要が生じる．これに対し，APSO では $\mathbf{g}_{best}$ を各粒子の評価が完了するごとに更新する．具体的には，Algorithm 9 に示すように，全体の流れは従来の PSO と同様であるが，グローバルベスト $\mathbf{g}_{best}$ を各粒子の評価ごとに更新している．また，APSO を並列型に拡張した方法として並列型 APSO（parallel APSO：PAPSO）が提案されている [19]．PAPSO は，Master-Slave 型の並列型 EA であり，各粒子の評価が並列に実行され，粒子の位置と速度，グローバルベスト $\mathbf{g}_{best}$ が並列非同期に更新される．これにより，各粒子は他の粒子の評価が完了する前にグローバルベストを更新し，探索を継続することが可能になる．しかし，(P)APSO は PSO を拡張した手法であるため実数値を扱う最適化以外の組合せ最適化や離散値の最適

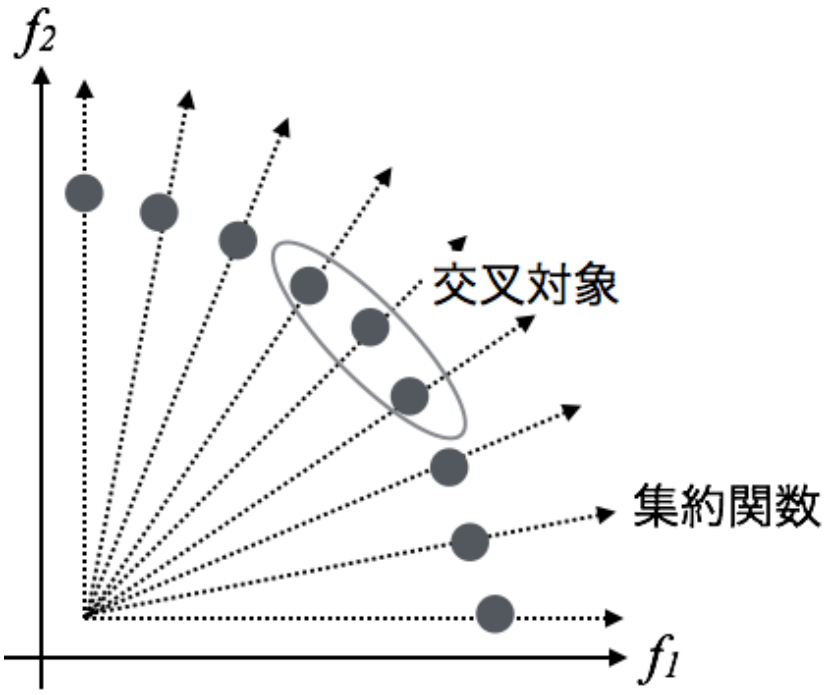


図 3.7　MOEA/D の概略図

Algorithm 9 APSO のアルゴリズム

1: 全粒子の位置 $\mathbf{x}_i$ と速度 $\mathbf{v}_i$ を初期化
2: $\mathbf{p}_{best}$ を現在位置で初期化
3: **for** $i = 1$ から粒子数分 **do**
4:　　i 番目の粒子を評価
5:　　i 番目の粒子の $\mathbf{p}_{best}$ を更新
6:　　$\mathbf{g}_{best}$ を更新
7:　　式 (3.3)，(3.4) に基づいて i 番目の粒子の位置と速度を更新
8: **end for**
9: 終了条件を満たさなければ 3. に戻る

化には適用が困難であるという問題がある．

3.2.5　非同期 steady-state GP

Maxwell は，SSGA を GP に適用した steady-state GP（SSGP）[32] を非同期に拡張した非同期 SSGP（asynchronous steady-state GP：ASSGP）を提案している [17]．具体的に，SSGA（SSGP）では全個体の評価が完了した段階でランダムに選択した解からトーナメント選択によって選択，削除を行うのに対し，ASSGP では図 3.8 に示すように

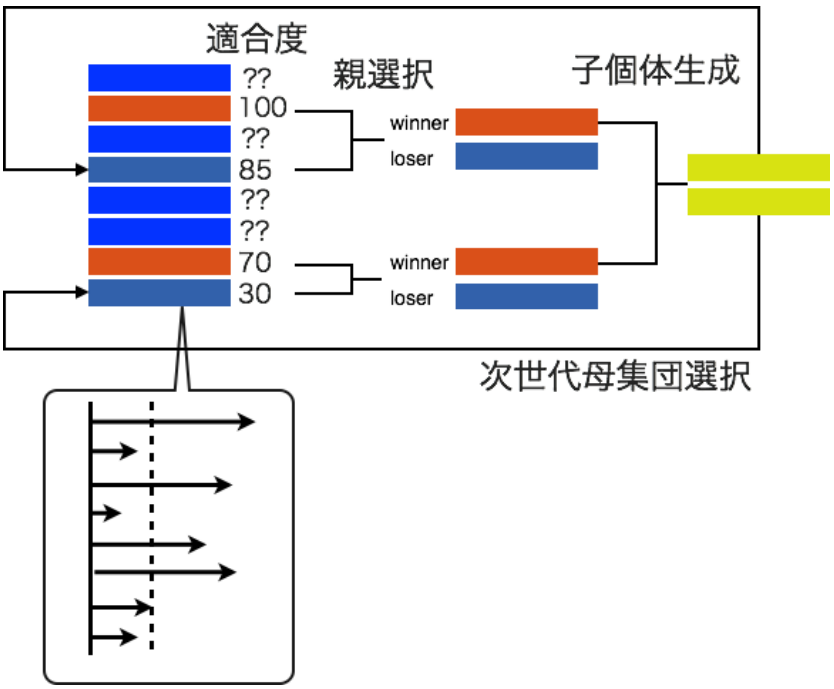


図 3.8　ASSGP の概略図

Algorithm 10 SSGA のアルゴリズム

1: 初期母集団を生成
2: 各個体を（擬似的に）並列に評価し，4 個体の評価を待機
3: 2. の 4 個体からトーナメント選択を 2 回行い，勝者を *winner*，敗者を *loser* とし，*winner* の 2 個体を親個体とする
4: 選択した親個体から遺伝的操作を通して 2 個の子個体を生成
5: *loser* を母集団から削除し，2 個の子個体を母集団に加える
6: 終了条件を満たさなければ 2. に戻る

各個体を（擬似的に）並列に評価し，4 個体の評価が完了するごとに評価が完了した個体からトーナメント選択により親選択，子個体生成が実行される．これにより，全個体の評価を待たずに 4 個体の評価が完了するごとに進化を継続することが可能になる．具体的な ASSGP の流れを Algorithm 10 に示す． ASSGP は，GP を非同期に扱う方法として提案されているが，このアルゴリズムは他の EA にも適用可能である．

3.3　本研究の位置づけ

上記の従来手法の特徴を評価，子個体生成，生存選択をそれぞれ同期か非同期かに分けて表 3.1 にまとめる．従来の同期 EA では，評価，子個体生成，生存選択を世代の概念を用いて同期して実行する．SSGA では，生存選択は他の個体の評価値によらずトーナメン

表 3.1　従来 EA の特徴と本研究の位置づけ

手法	同期 EA	SSGA	PEA	DE PSO	MOEA/D	(P)APSO ASSGP 本研究
評価	同期	同期	非同期	同期/非同期	同期/非同期	非同期
子個体生成	同期	同期	同期	同期	非同期	非同期
生存選択	同期	非同期	同期	非同期	同期	非同期

ト選択に選ばれた個体のみで非同期に実行できるが，評価，子個体生成は同期して実行する．PEA では，評価を非同期（並列）に実行し，子個体生成，生存選択は同期 EA と同様に世代の概念を用いて同期して実行する．DE では，評価は一般的には同期であるが非同期にも実行可能であり，生存選択は親個体と子個体の間で非同期に実行可能であるが，子個体生成には全体情報が必要になる場合があるため同期して実行する．MOEA/D では，評価は一般的には同期であるが非同期にも実行可能であり，子個体生成は近傍の集約関数ベクトルのみを用いて非同期に実行可能であるが，生存選択の際にすべての集約関数を考慮する必要があるため同期に実行する．(P)APSO と ASSGP は，評価を非同期（並列）に実行し，評価が完了した個体から非同期に子個体生成，生存選択を実行する．

これらの手法のうち，(P)APSO，ASSGP を除く手法は，評価，子個体生成，生存選択のいずれかを全体の評価に基づいて実行する必要があるため評価時間差のある問題に対して十分に対処できない．また，(P)APSO，ASSGP のいずれの手法も評価の完了しない個体に対しては従来の同期進化と同様に待機時間の上限を与えることで対処しており，同期進化と同様の問題を抱えている．さらに，従来の非同期 EA である (P)APSO は，実数値ベクトルに基づいて子個体を生成する PSO をもとにした実数値の最適化に特化した手法であり，組合せ最適化や経路最適化のような問題に対処できない．

上記の問題に対し，本研究では評価，子個体生成，生存選択をすべて非同期に実行し，(1) 部分的な評価情報のみを用いて全体の評価情報を利用可能な同期 EA を上回る性能を有し，かつ (2) 評価が完了しない解が存在する場合でも対処可能な問題クラスに依存しない新しい非同期 **EA** フレームワークを提案する．

3.4　研究のアプローチ

非同期進化の実現に向けては，部分的な評価のみを用いて適切に親個体選択と生存選択をすること重要になる．特に，非同期進化では評価が完了した段階でその個体を親個体として選択するか淘汰するかを決定するため，ASSGP のように評価が完了した 2 個体を比較して選択，淘汰を決定する方法では評価時間は短いが評価値の低い個体が親として選択

される頻度が高くなり，進化の妨げになる可能性がある．この課題を解決する非同期 EA フレームワークとして，本研究では，(1) 部分的な情報を事前に定める絶対的な基準を前提に絶対評価にもとづいて評価する方法（絶対評価に基づく非同期 EA）と (2) 部分情報間の相対的な比較にもとづいて評価する方法（相対評価に基づく非同期 EA）を提案する．ここで，絶対評価とは個体の適合度そのものを用いて親選択をする方法であり，実際の評価値が重要になる．これに対し，相対評価は適合度同士の大小関係に基づいて親選択をする方法であり，実際の値は親選択に関係しない．(1) 絶対評価に基づく非同期 EA としては，自己複製するデジタル生物の非同期な進化をシミュレートする Tierra[22] を拡張し，絶対評価にもとづいて解を進化させる Tierra 型非同期 EA（Tierra-based Asynchronous TAEA：TAEA）を提案する．(2) 相対評価に基づく非同期 EA としては，TAEA を拡張し，探索中に得られた優良個体を保持するアーカイブとそこから選択されるリファレンス個体の情報を解探索の指標として相対評価にもとづいて解を進化させる非同期リファレンス評価を用いる EA（Asynchronous Reference-based Evaluation EA：ARE-EA）を提案する．

図 3.9，3.10，3.10 に従来の同期 EA，本研究で提案する (1) 絶対評価に基づく非同期 EA，および (2) 相対評価に基づく非同期 EA の概念図を示す．図 3.9，3.10，3.10 において，各円は個体を表し，濃灰色が評価済み個体，薄灰色が未評価個体，黄色星印は評価済み個体の中での優良個体をそれぞれ表す．まず，図 3.9 に示す従来の同期 EA は評価済みの全個体の評価値をもとに優良個体の情報を利用して次世代の解集団を生成する処理を繰り返すことによって解を進化する．これに対し，図 3.10 に示す (1) 絶対評価に基づく非同期 EA では，解探索において解の優劣の基準（図中赤円）を事前に定め，評価済みの個体をその基準をもとにして進化する．これにより，全個体の評価を必要とせず，各個体の評価値と事前に定めた基準のみに基づいて非同期に進化をすることが可能となる．一方，図 3.11 に示す (2) 相対評価に基づく非同期 EA では，(1) 絶対評価に基づく非同期 EA とは異なり事前の基準は設けず，進化の過程において評価済みの個体の情報に基づいて解の優劣の基準を更新し，それに基づいて解を進化する．これにより，(1) 絶対評価に基づく非同期 EA と同様に全個体の評価を必要とせずに非同期に進化をすることが可能となる．このように，本研究で提案する非同期 EA では，評価が完了した個体の評価情報以外の基準を用意し，その基準に基づいて親個体選択と生存選択を行うため，評価時間が短く評価値の低い個体の選択頻度が高くなることを抑制する．

本研究が対象とする問題領域を図 3.12 にまとめる．本研究では，TAEA と ARE-EA の有効性を検証するために GP の例題を用いた実験を行う．GP の例題として，はじめに (1) 与えられた目的を達成可能な完全なプログラムの進化によって最適化する問題を扱い，続いて (2) ランダムに生成されたプログラムを進化させ目的達成に近づける問題を扱う．(1) 与えられた目的を達成可能な完全なプログラムの進化では，実応用を想定した

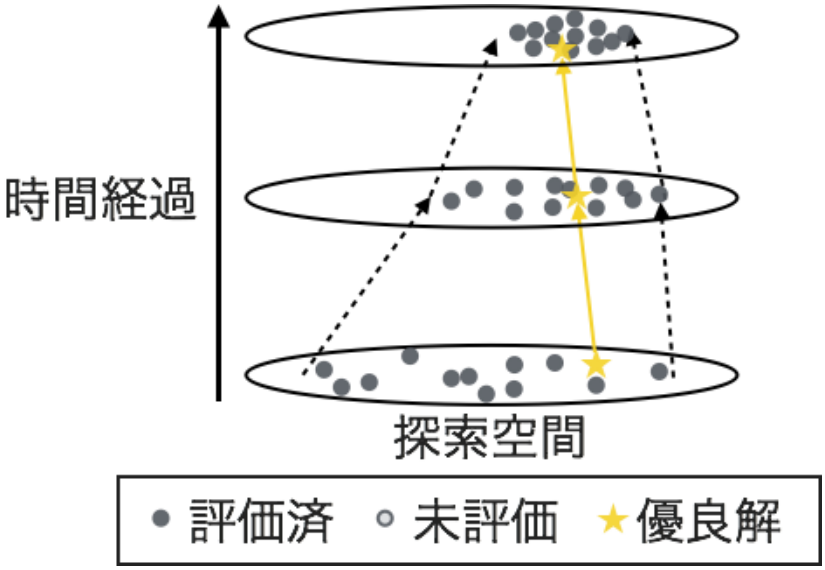


図 3.9　同期 EA の概念図

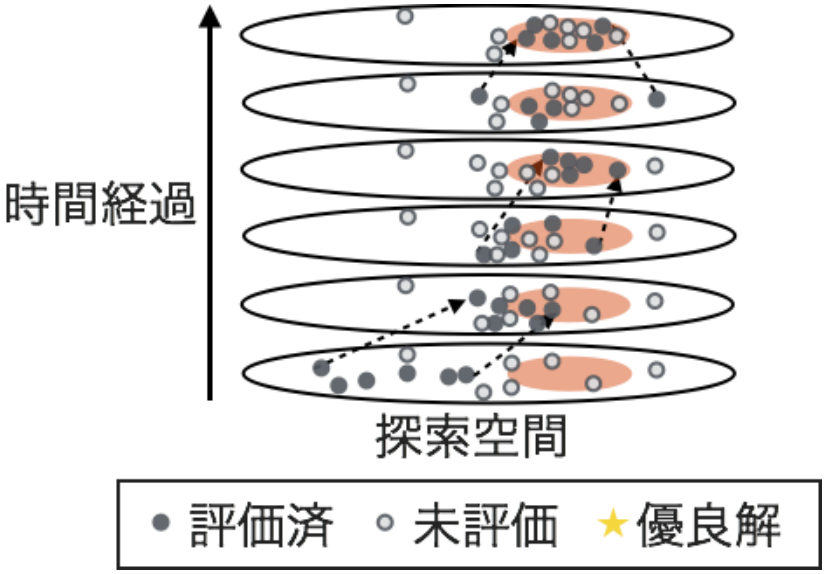


図 3.10　(1) 絶対評価に基づく非同期 EA の概念図

PIC マイコンで実行可能な命令を含むプログラムの進化を扱う．(2) ランダムに生成されたプログラムの進化では，GP の一般的な例題である関数同定問題（symbolic regression problem）を扱う．また，非同期 EA の宇宙機への適用を念頭においた (3) 宇宙環境においてプログラムにビット反転が発生する環境下におけるプログラム進化の問題を扱う．

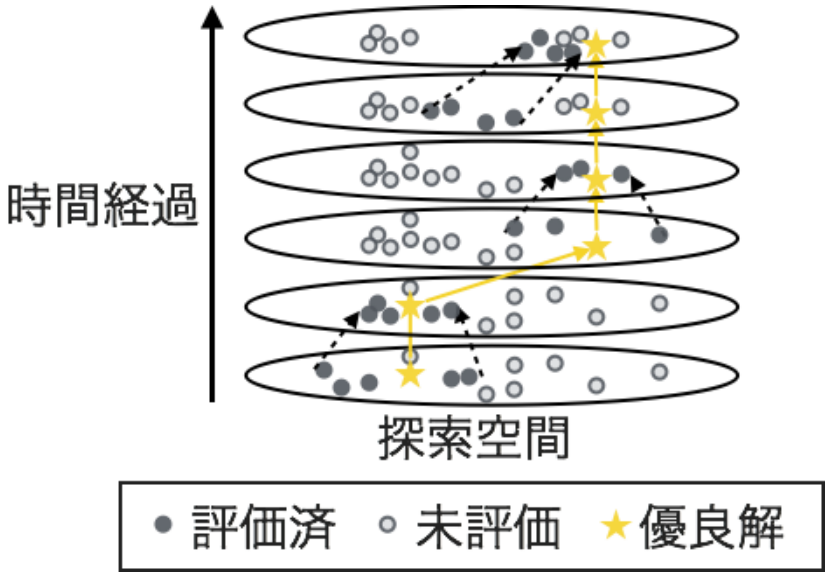


図 3.11　(2) 相対評価に基づく非同期 EA の概念図

非同期EAの種類 ＼ 問題の種類		プログラム		ランダムな関数
		ビット反転なし	ビット反転あり	
評価方法	絶対評価 TAEA	本研究の対象領域		
	相対評価 ARE-EA			

図 3.12　対象とする問題領域

第 4 章

Tierra 型非同期進化的アルゴリズム（Tierra-based Asynchronous Evolutionary Algorithm：TAEA）

4.1 概要

本章では，絶対評価に基づく非同期進化手法として，Tierra 型非同期進化的アルゴリズム（*Tierra-based Asynchronous Evolutionary Algorithm: TAEA*）を提案する．Tierra は非同期にプログラム（デジタル生物）を進化可能であるという特徴に着目し，Tierra で自己複製プログラム以外の個体を進化させるために適合度（*fitness*）の概念を導入し，適合度に基づく複製，削除のメカニズムを可能にする．以下では，はじめに TAEA のもととなる Tierra について説明し，その後，本章で提案する TAEA のアルゴリズムについて説明する．

4.2 Tierra

Tierra [33, 22, 34] は T. S. Ray によって提案されたデジタル生物の進化シミュレータである．Tierra の概略図を図 4.1 に示す．Tierra は地球の生態系を模擬して設計されており，デジタル生物がコンピュータプログラム，空間がメモリ，活動エネルギーが CPU 時間にそれぞれ相当する．デジタル生物は限られたメモリ（実生物の地球に相当）内に生息し，自分自身をメモリ内の空き領域に自己複製（コピー）することで子孫を残すことを目的とする．デジタル生物はアセンブリ言語プログラムのように一次元配列上の命令列によって記述される．複製の際には EA などと同様の交叉や突然変異，または命令語の挿入，削除などの遺伝的操作によってプログラムの一部が変更される．各生物には活動エネルギーに相当する CPU 時間が割り当てられ，生物は与えられた時間の中でプログラムを

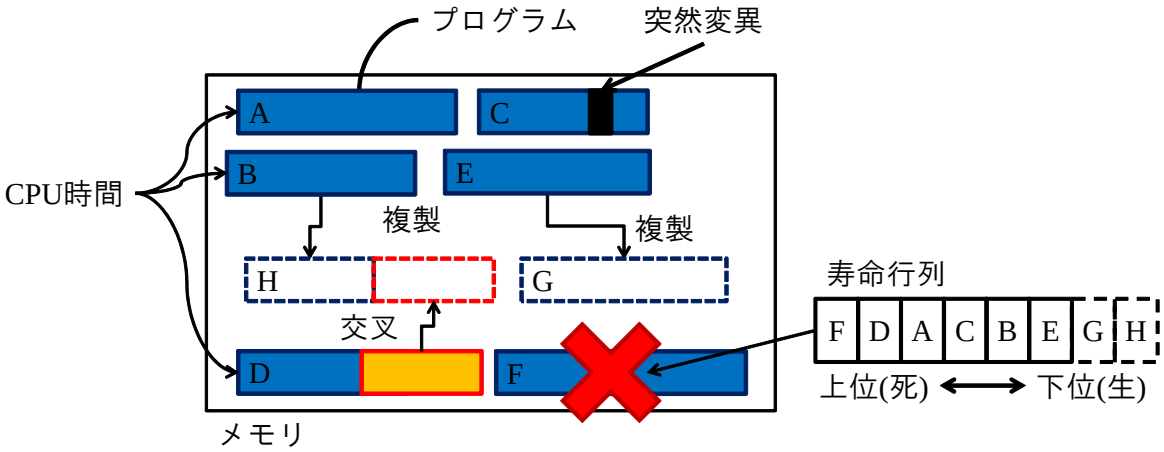


図 4.1　Tierra の概略図

実行する．与えられる CPU 時間を各生物のプログラムに対して十分短く設定することによって，擬似的にプログラムの並列実行を実現している．デジタル生物はリーパーキュー（*reaper queue*）と呼ばれる行列に並べられ，行列内の順番によって寿命を制御される．各生物はプログラムの実行が成功すれば行列内の位置が一つ下がり，失敗すれば位置が一つ上がる．そして，メモリ内がプログラムで満たされると，ある閾値を超えた空き領域ができるまで行列の上位から生物が削除される．リーパーキューの機能により，プログラムを正しく実行できる生物は長く生き延び，逆に失敗の多い生物は削除されやすくなる．Tierra の大まかな流れは下記の通りである：

1. デジタル生物が割り当てられた CPU 時間内で命令を実行
2. 命令を正しく実行できた生物はリーパーキュー内の下位に移動し（寿命が延びる），実行に失敗すれば上位に移動する（寿命が縮む）
3. 命令実行が完了し，複製された生物はリーパーキューの最下位（最も寿命が長い）に追加される
4. メモリが生物で満たされた場合，リーパーキューの上位の生物から順に削除する
5. 1 に戻る

Tierra の大きな特徴として，突然変異によるプログラムの進化が挙げられる．例えば，別のプログラムの自己複製部分を利用することによって自己複製する寄生種 (*parasite*) と呼ばれるプログラムや，その寄生種に対して免疫を持ち，寄生種に自己のプログラムを複製させる超寄生種 (*hyper-parasite*) と呼ばれるプログラムが進化によって生成される [33][22][34]．寄生種は図 4.2 に示すように先祖種の内，自己複製ループ部が自分のコピープロシージャーではなく他のプログラムのコピープロシージャーを呼び出すように変化したプログラムであり，先祖種よりもコピープロシージャーを持たない分サイズが小さく

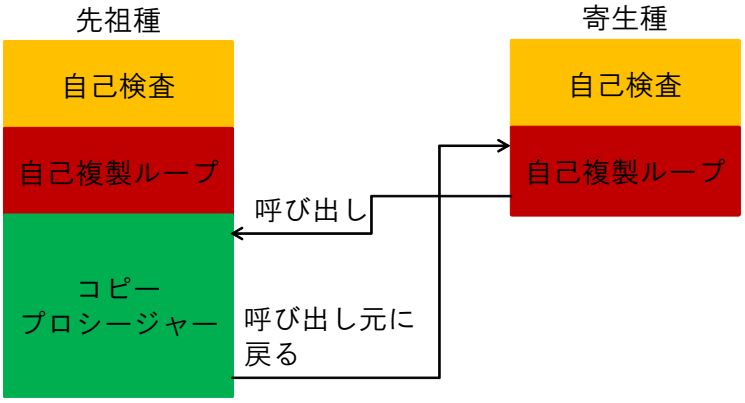


図 4.2　Tierra における寄生種

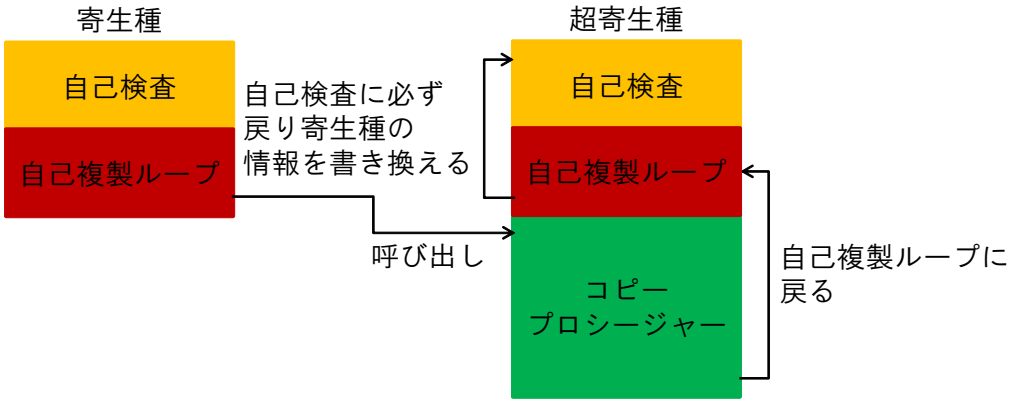


図 4.3　Tierra における超寄生種

効率的に複製が可能になる．これに対し，超寄生種は図 4.3 に示すようにコピープロシージャーの実行終了後に必ず自己検査からやり直す過程を踏むため，寄生種からコピープロシージャーを呼び出された場合に寄生種の情報 (コピーするプログラムの先頭と末尾) を自己検査部で書き換えることで寄生種に超寄生種自身のプログラムを複製させる．

また，単独では複製できないが，2 つのプログラムがお互いに利用しあうことで初めて複製できるようになるという，共生の関係を持つ共生種も報告されている [34]．さらに先祖種のプログラムよりもサイズが小さくなることで複製に要する時間を減らすように進化したプログラムや，効率のよいアルゴリズムによって短い時間で複製できるように進化したプログラムも確認されている [35]．具体的には，22 個の命令だけで自己複製が可能になるプログラムや，コピープロシージャーのループを展開することで終了判定やジャンプ処理を減らし，複製にかかる命令実行回数を削減するように進化したプログラムが確認されている．

このような進化はあらかじめ Tierra に組み込まれているものではなく，繰り返される突然変異の結果として起きる創発 (emergence)[36] によって生まれるものである．

4.3 TAEA

TAEA は非同期進化に基づく EA であり，前節で示した Tierra[22] の設計をもとに，任意の個体を進化させるために適合度（*fitness*）を導入し，適合度に基づいた複製，淘汰の機構を導入した．

図 4.4 に TAEA の概念図を示す．全個体はリーパーキュー[*1]と呼ばれる行列に記憶される．各個体は（擬似的に）並列に評価され，評価が完了するごとに各個体の適合度を算出する（図 4.4 中 Step1）．適合度がある閾値を超えると，その個体が親個体として選択され，交叉や突然変異などの遺伝的操作を通して子個体が生成される（図 4.4 中 Step2）．生成された子個体はリーパーキュー内の最下位の最も削除から遠い位置に追加される（図 4.4 中 Step3）．例えば，図 4.4 において，ind_1 と ind_3 の適合度が閾値を超えた場合，これらを親個体として子個体が生成される．それに対し，ind_2 の適合度が閾値を超えない場合，親個体として選択されない．親選択の際に集団内に優良個体を保持するため，適合度が最大であり，かつ直前に適合度が最大であった個体よりも優れている場合，その個体はエリートとして遺伝的操作なしに同じ遺伝子を持つ個体がそのまま複製される．例えば，図 4.4 において，ind_3 が最大適合度を持つ場合，ind_3 はエリート個体として複製される．リーパーキュー内の順位は，各個体の適合度が閾値を超えたかどうかによって変動する．もし閾値を超えていればその個体の位置はリーパーキューの下位に移動し，超えていなければ上位に移動する（図 4.4 中 Step4）．例えば，図 4.4 において，ind_1 と ind_3 は適合度が閾値を超えているためリーパーキューの下位に移動し，ind_2 は閾値を超えていないため上位に移動する．子個体の生成によって集団内の個体数が集団サイズを超過した場合，リーパーキューの上位の個体から順に削除される（図 4.4 中 Step5）．例えば図 4.4 では，子個体が新たに 3 個体生成されるため，リーパーキューの上位 3 個体が削除される．

以下に TAEA の選択と寿命の制御，子個体生成，削除の各手順について詳細を示す．

4.3.1 選択と寿命の制御

ある個体の評価が完了した場合，その個体の適合度を算出する（図 4.4 中 Step1）．各個体はそれぞれ適合度を累積適合度（f_{acc}）として累積し，その累積値をもとに選択，寿命の制御が行われる．具体的には，最大適合度を f_{max} と表すと，累積適合度が f_{max} を超える個体は親個体として選択され，選択された個体は累積適合度から f_{max} が差し引かれ

[*1] ただし，ここで「キュー」は FIFO 型の「キュー（Queue)」ではなく，削除の優先度順に並んだ「プライオリティキュー」の意味で用いる．

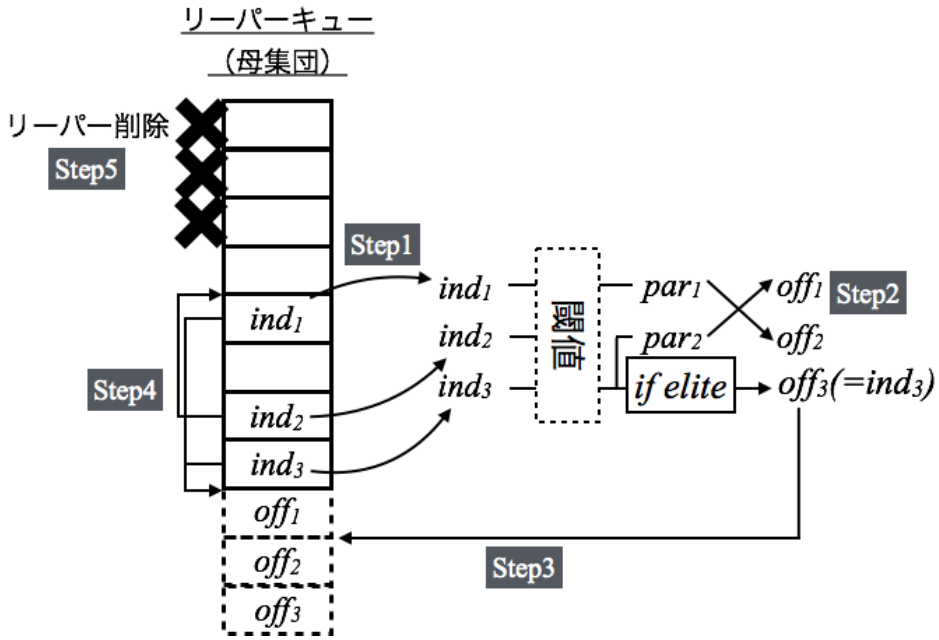


図 4.4　Tierra 型非同期進化的アルゴリズム（TAEA）の概略図

る．この選択条件により，最良個体の適合度を f_{max}，そこから差が大きくなるほど適合度が減少するように適合度関数を設計することで適合度が最大の個体は必ず親個体として選択され，それ以外の個体は適合度が高いほど累積値が最大適合度を超える頻度が高くなるため選択されやすくなる．

寿命の制御は適合度評価の際のリーパーキューの制御によって実現される（図 4.4 中 Step4）．親個体として選択された個体はリーパーキュー内の位置が下げられ，生き残りやすくなり，逆に選択されない個体は位置が上げられ，削除されやすくなる．具体的に，行列内での移動量は適合度に基づいて下記の式で算出される確率によって決定する．

$$P_{down}(f) = \frac{f}{f_{max}} \times \frac{N-1}{N} \tag{4.1}$$

$$P_{up}(f) = \frac{f_{max}-f}{f_{max}} \times \frac{N-1}{N} \tag{4.2}$$

式 (4.1)，(4.2) において N はリーパーキュー内の個体数を表す．式 (4.1)，(4.2) で算出された確率をもとに Algorithm 11 に示す手順でリーパーキュー内の位置が制御される．Algorithm 11 において，$rand(0,1)$ は 0 から 1 の実数値の乱数を表し，$ind.f$ は個体 ind の適合度を表す．この制御により，行列内の位置が下る場合には適合度が高いほどより行列内の下位まで位置が下がり，逆に上がる場合には低いほど行列内の上位まで上がるようになる．

例として，表 4.1 に示すような適合度が高い場合（f_{max}）と低い場合（$f_{max}/10$），評価時間が短い場合（評価時間 1 単位時間）と長い場合（評価時間 5 単位時間）の組み合

Algorithm 11 TAEA におけるリーパーキュー制御のアルゴリズム

1: **repeat**
2: 　リーパーキュー内の位置を一つ下げる（上げる）
3: **until** $rand(0,1) < P_{down(up)}(ind.f)$

表 4.1　適合度と評価時間の組み合わせ

適合度 / 評価時間	高い（**H**igh）$f = f_{max} = 100$	低い（**L**ow）$f = f_{max}/10 = 10$
短い（**F**ast）$t = 1$ 単位時間	H-F	L-F
長い（**S**low）$t = 5$ 単位時間	H-S	L-S

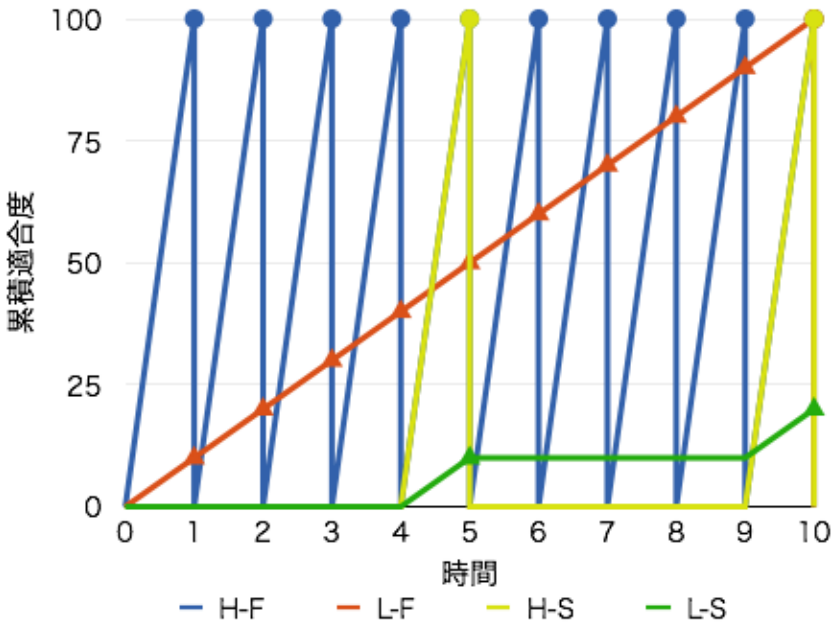


図 4.5　適合度と評価時間の関係による時間ごとの累積適合度の推移

わせで 4 つの個体を想定した場合，これらの累積適合度の変化と親選択回数は図 4.5 のようになる．ここで，表 4.1，および図 4.5 において，H は適合度が高い場合（**H**igh），L は適合度が低い場合（**L**ow），F は評価時間が短い場合（**F**ast），S は評価時間が長い場合（**S**low）をそれぞれ表し，それぞれのアルファベットを組み合わせて表現する（例えば，適合度が高く，評価時間が短い場合は H-F と表記）． 図 4.5 において横軸は時間，縦軸はそれぞれの場合の累積適合度の推移を表し，色の違いはそれぞれ 4 種類の個体を表す．丸印，三角印で記した時間に評価が完了し，丸は累積適合度が $f_{max}(= 100)$ を超えて親と

して選択されることを表し，三角は累積適合度が f_{max} を超えず親として選択されないことを表す．まず，青線で示した評価時間が短く適合度の高い個体（H-F）は 1 単位時間毎に累積適合度が f_{max} に達し，頻繁に親として選択される（図中青丸）．それに対し，黄線で示した同一の適合度を持つが評価時間の短い個体（H-S）は親選択頻度は少ないものの，評価が完了した際には必ず親として選択されることがわかる（図中黄丸）．一方，赤線で示した評価時間は短いが適合度の低い個体（L-F）は評価が完了し，累積適合度の値は徐々に増加していくものの，増加量が少ないため結果として H-S よりも親選択頻度は少なくなる（図中赤丸）．また，評価完了時に累積適合度が閾値を超えないため，リーパーキュー内での位置が上位に移動する頻度が多く（図中赤三角），削除の対象となりやすい．最後に，緑線で示した評価時間が長く適合度の低い個体（L − S）は累積適合度が他の個体と比較して増加量が非常に少ないため親選択の機会を得ることができない．また，生成された子個体や閾値を超えてリーパーキューの下位に移動する個体との相対的な位置関係で徐々に上位に移動するため，相対的に削除されやすくなる．以上の例のように，TAEA の累積適合度と閾値に基づく親選択と寿命の制御によって，適合度の高い個体は親個体として選択されやすく，適合度の低い個体は親個体として選択されにくくなる．特に，評価時間が短く適合度の低い個体はリーパー制御によって削除の候補となりやすくなる．

TD トーナメント選択

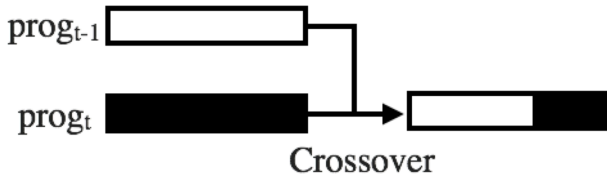
TAEA では，常に優良個体が親として選択される保証がないという問題がある．これは，評価が短時間で完了する個体ほど累積適合度が高くなりやすく，適合度が低い場合でも親として選択される頻度が多くなるためである．この問題を解決するために，非同期に優良個体を選択可能な方法として TD（temporal difference）トーナメント選択（TDTS）を提案する．

TDTS は累積適合度が閾値を超えた直前の λ 個体と現在の個体を比較し，優れた個体を親個体として選択する．また交叉の際には，直前に親個体として選択された個体と現在の親個体を組み合わせて子個体を生成する．ここで，λ はどれくらい前の個体までを比較対象とするかを決定するパラメータである．$\lambda = 0$ の場合は TDTS なしと同一の比較なしの選択であり，λ が大きくなるほどより過去の個体までを比較対象とするようになる．TDTS のアルゴリズムを Algorithm 12 に示す．

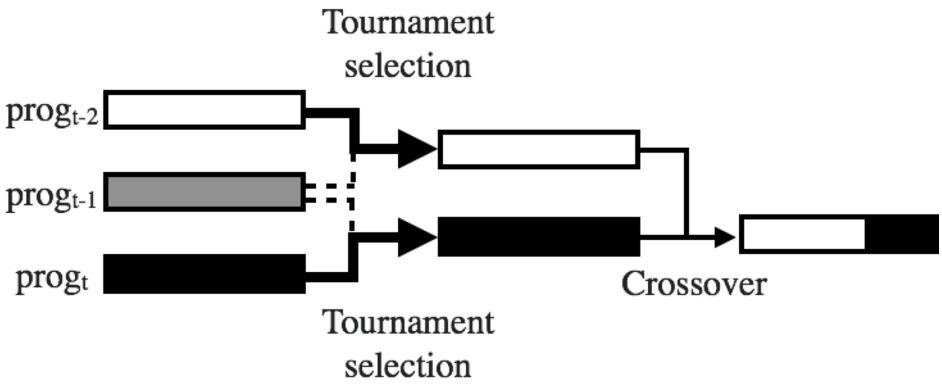
図 4.6 に従来の親選択法と $\lambda = 1$ の場合の TDTS の概略図を示す．従来の選択法と TDTS の差異は，従来の選択法では評価が完了した個体は累積適合度が閾値を超えていればそのまま親個体として選択されるが，TDTS では過去の個体との比較によって親個体が選択される点である．そのため，TDTS において適合度は高くないが評価時間が短いために閾値を超える個体は親個体の候補となることができるが過去の個体との比較から親個体として選択されなくなる．以降，TDTS を導入した TAEA を TAEA/TDTS と表す．

Algorithm 12 TDTS のアルゴリズム

1: $T \cup \{ind_t\}$ から最も優れた個体を親個体（$winner_t$）として選択
2: 遺伝的操作を行い現在の親個体（$winner_t$）と直前の親個体（$winner_{t-1}$）から子個体を生成
3: $T \leftarrow T \cup \{ind_t\} \setminus ind_{t-\lambda}$
4: $winner_{t-1} \leftarrow winner_t$



(a) TAEA における従来の親選択法



(b) $\lambda = 1$ の TDTS による親選択

図 4.6　従来の選択法と $\lambda = 1$ の場合の TDTS の概略図

4.3.2　子個体生成

TAEA では，親個体として選択された個体をもとに一定の確率で交叉，突然変異，命令語の挿入・削除の遺伝的操作を実行し，新しい子個体を生成する．交叉は，直前に親個体として選択された個体との間で実行され，交叉点は命令語単位で 2 個体独立に決定する．突然変異は任意の一命令をランダムに変化させる．命令語の挿入・削除は複製される個体の任意の位置に命令語を挿入，または削除する．さらに，最大適合度を持つ個体を保持するために，エリート保存戦略 [37] を導入する．具体的には，現在の個体の適合度が最大適合度以上の場合，その個体をエリートの候補とする．そして，直前にエリートの候補となった個体と比較し，優れている場合は遺伝的操作を行わず個体を複製（コピー）し，劣っている場合は遺伝的操作を適用して複製する．

4.3.3 削除

TAEA では，集団（リーパーキュー）内の個体数が最大集団サイズを超過する場合，最大集団サイズ以下になるまでリーパーキューの上位数 % （例えば 30%）からランダムに個体を削除する．リーパーキューに基づく削除では長い間生き残った個体や適合度の低い個体が削除される．

4.3.4 アルゴリズム

TAEA のアルゴリズムを Algorithm 13 に示す．Algorithm 13 において，ind は評価が完了した個体，$ind.f$，$ind.f_{acc}$ はそれぞれ ind の適合度と累積適合度，MAX_POP は最大集団サイズを示す．ind_{elite} は直前に適合度（$ind.f$）が最大適合度（f_{max}）であった個体を表す．各個体は（擬似的に）並列に評価され（1 行目），評価が完了するごとに各個体は適合度を $ind.f_{acc}$ に累積する（2 行目）．累積適合度が閾値（Algorithm 13 では最大適合度 f_{max}）を超えると，その個体が親個体として選択され（3 行目），累積適合度から閾値が差し引かれる（4 行目）．逆に，閾値を超えない個体は Algorithm 11 に基づいてリーパーキューの上位に移動する（19 行目）．親個体として選択された個体は Algorithm 11 に基づいてリーパーキューの下位に移動し（5 行目），子個体を生成する（6 行目）．$ind.f$ が最大適合度であり，かつ直前に適合度が最大であった個体（ind_{elite}）よりも優れている場合にはその個体はエリートとして遺伝的操作なしに同じ遺伝子を持つ個体がそのまま複製され（9 行目），劣る場合には遺伝的操作を適用して子個体を生成する（11 行目）．集団内の個体数が集団サイズ MAX_POP を超過した場合，リーパーキューの上位数 % の個体からランダムに削除する（15 行目から 17 行目）．以上の操作を繰り返すことで TAEA は非同期に個体を進化させる．

4.4 適応型 TAEA（TAEA^{+}）

4.3 章で示した TAEA(/TDTS) の課題として，以下の 2 点がある．

1. 適合度関数の設計によって累積適合度と閾値に基づく親個体選択が影響を受けるため，問題の性質が未知の問題に対して適用が困難である，具体的には，(1) 複製条件となる閾値（f_{max}）の設定と (2) 累積適合度の累積度合い（*i.e.*，適合度関数の傾き）の適切な設定が求められる．
2. リーパー制御パラメータ P_{down} の値によって集団内の多様性が大きく左右されるため，P_{down} の設計が解探索性能に影響を与える．具体的には，集団内に優良個体を保持するために P_{down} は高く設定すべきであるが，優良個体が多数集団内に存

Algorithm 13 TAEA のアルゴリズム

```
1: ind の適合度（ind.f）を計算
2: ind.f_acc ← ind.f_acc + ind.f
3: if ind.f_acc ≥ f_max then
4:    ind.f_acc ← ind.f_acc − f_max
5:    ind のリーパーキュー内の位置を下位へ移動
6:    遺伝的操作を行い ind の子個体を生成
7:    if ind.f ≥ f_max then
8:       if ind > ind_elite then
9:          ind を遺伝的操作なしで複製
10:      else
11:         ind を遺伝的操作ありで複製
12:      end if
13:      ind_elite ← ind
14:   end if
15:   if 集団内の個体数 >MAX_POP then
16:      リーパーキューの最上位個体から削除
17:   end if
18: else
19:   ind のリーパーキュー内の位置を上位へ移動
20: end if
```

在する場合には逆に多様性を失う原因となるため，解探索を促進するためには優良個体の保持と多様性の維持のバランスを保つパラメータ設定が必要となる．

本研究では，この 2 つの課題を解決するために 4.3 で示した TAEA/TDTS を基礎として，(1) 適合度関数の自動調整，(2) リーパー制御パラメータ P_{down} の自動調整を加えて拡張した適応型 TAEA（TAEA$^+$）を提案する．まず，4.4.1 章で (1) 適合度関数の自動調整として四分位数に基づく適合度スケーリング（Quartile based Fitness Scaling：QFS）について説明し，続いて 4.4.2 章で (2)P_{down} の自動調整としてエリート割合に基づく適応的 P_{down} を説明する．以降，これら 2 つの方法を加えた TAEA/TDTS を TAEA$^+$ と記述し，特に断りが無い限り最小化問題を用いて説明する．

4.4.1 四分位数に基づく適合度スケーリング（Quartile based Fitness Scaling：QFS）

概要

TAEA では，累積適合度と閾値を用いて絶対評価に基づいて個体の選択，削除を実現するため，適合度関数の設計が重要になる．具体的に最小化問題において，個体の評価値 f を最大適合度 f_{max}，最小適合度 0 にスケーリングする線形の適合度関数 $s(f)$ を

$$s(f) = \begin{cases} f_{max} & if \quad f < f_1 \\ f_{max} \times \dfrac{f_0 - f}{f_0 - f_1} & if \quad f_1 \leq f \leq f_0 \\ 0 & otherwise \end{cases} \tag{4.3}$$

と定義する場合，f_0 と f_1 の値によって累積適合度の累積度合いが変化するため，f_0 と f_1 を適切に設計することが必要となる．しかし，一般に対象とする最適化問題の性質は未知であるため，事前に適切な適合度関数を設計することは困難である．

EA において適合度スケーリングをする場合，集団内の最大値と最小値を用いる方法が一般的である．しかし，非同期進化では一部の個体の評価値のみを用いるため集団内の最大値と最小値を求めることができない．また，TAEA では適合度の絶対値に基づいて選択，削除を決定する絶対評価を用いるため，外れ値を含む場合に評価値に対する適合度が過大評価される可能性がある．そこで本研究では，上記の問題を解決する適合度スケーリングとして非同期に得られる評価値のサンプルから四分位数に基づいて f_0 と f_1 を設定する方法を提案する．

四分位数

四分位数（quartile）とは，数値集合の代表値を表す方法の一つで，昇順に並んだ n 個の数値集合 $x_i(i \in \{1, \cdots, n\}, x_1 \leq x_2 \leq \cdots \leq x_n)$ のうち，数値集合を 4 等分する区切りの上位 1/4 番目，2/4(= 1/2) 番目，3/4 番目の数値を指し，それぞれ第 1 四分位数 ($Q1$ と表記)，第 2 四分位数（または一般に中央値)，第 3 四分位数（$Q3$ と表記）と呼ぶ．四分位数は図 4.7 に示すような箱ひげ図で表すことができ，$Q1$ と $Q3$ を箱の両端としてそこから最大値と最小値に向けてひげを伸ばして記される．具体的に，図 4.7 では x_1 から x_9 の 9 つのデータに対し，1/4 番目 $= x_3$，1/2 番目 $= x_5$，3/4 番目 $= x_7$ がそれぞれ $Q1$，中央値，$Q3$ となる．また，集合のばらつきを表す値として $Q3$ と $Q1$ の差 $Q3 - Q1$ で表される四分位範囲（IQR と表記）を用い，一般的に $Q3 + 3.0 \times IQR$ よりも大きな値は極度の外れ値とみなされる．

四分位数は平均値と分散を用いる方法と比べて外れ値を含む集合にも代表値への影響が少なく頑健なため，TAEA の適合度スケーリングのパラメータを決定する指標として適

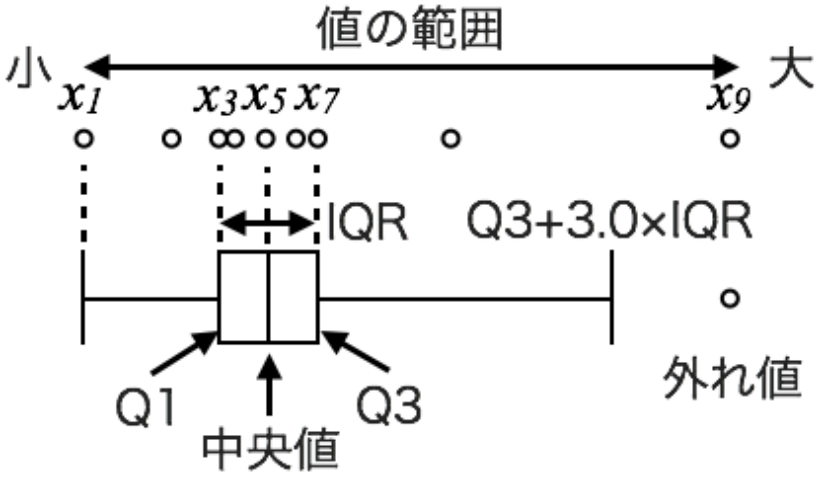


図 4.7　四分位数の箱ひげ図による表現

している．

QFS のアルゴリズム

四分位数に基づく適合度スケーリング（Quartile based Fitness Scaling：QFS）のアルゴリズムを Algorithm 14，その概略図を図 4.8 に示す．Algorithm 14 において $list_f$ は評価値を蓄積する配列，$list_f.Min$，$list_f.Q3$，$list_f.IQR$ はそれぞれ $list_f$ の最小値，第 3 四分位数（$Q3$），四分位範囲（$IQR = Q3 - Q1$）を表し，$min(x, y)$ は x と y のうち小さい値を返す関数を表す．QFS では，非同期な進化の過程で評価が完了した順に一定数（Algorithm 14 中 N_f）の評価値を配列（Algorithm 14 中 $list_f$）に蓄積し（Algorithm 14 中 1 行目から 2 行目），その評価値集合の四分位数に基づいて式 (4.3) の f_1 に $list_f$ の最小値，f_0 に極度の外れ値の閾値となる $Q3 + 3.0 \times IQR$ を割り当てる（Algorithm 14 中 3 行目から 13 行目）．具体的に，f_1 は $list_f$ 中の最小値と現在の f_1 のうち小さい値（Algorithm 14 中 3 行目），f_0 は $list_f$ から求められる外れ値の閾値（$list_f.Q3 + 3.0 \times list_f.IQR$）のうち小さい値（4 行目から 8 行目）にそれぞれ設定する．ただし，f_1 と f_0 が同値になる場合には，式 (4.3)（$s(f)$ ）の分母が 0 になることを防ぐために f_0 は更新せず直前の値を使用する（5 行目）．また，f_1 が一定回数（Algorithm 14 中 N_{update}）更新されない場合，f_1 の値を現在の最小値（$list_f.Min$）に設定する（Algorithm 14 中 9 行目から 16 行目）．以降，設定した f_0 と f_1 を用いて算出された適合度を用いて同様の操作を行う．

Algorithm 14 四分位数に基づく適合度スケーリング（Quartile based Fitness Scaling：QFS）のアルゴリズム

$list_f$ に $ind.f$ を追加
if $|list_f| = N_f$ **then**
  $f_1 = min(f_1, list_f.Min)$
  $pref\_f_0 = f_0$
  $f_0 = min(f_0, list_f.Q3 + 3.0 \times list_f.IQR)$
  if $|f_1 - f_0| = 0$ **then**
    $f_0 = prev\_f_0$
  end if
  if $f_1 \geq list_f.Min$ **then**
    $f_1\_not\_updated = 0$
  else
    $f_1\_not\_updated++$
    if $f_1\_not\_updated \geq N_{update}$ **then**
      $f_1 = list_f.Min$
    end if
  end if
  $list_f = \emptyset$
end if

4.4.2 リーパー制御パラメータ P_{down} の適応的調整

リーパーキューの制御は式 (4.1)，および Algorithm 11 に基づいて実行され，リーパーキュー内の位置を下げる制御では適合度が高いほどよりリーパーキューの下位への移動量が多くなる（生き残りやすくなる）．これは，集団（リーパーキュー）内に優良個体を保持するためには必要であるものの，集団内の多様性を失う原因ともなる．そこで本研究では，QFS でサンプルした評価値（Algorithm 14 中 $list_f$）の分布に基づいて P_{down} を適応的に調整する方法を提案する．以降，適応的 P_{down} を P^{α}_{down} と表す．

P^{α}_{down} は $list_f$ 内の最小値（$list_f.Min$）と同値の要素の個数を n とする時

$$P^{\alpha}_{down} = 1.0 - \frac{n}{|list_f|} \tag{4.4}$$

として求められる．これにより，サンプルされた評価値の中で最小の評価値と同値の評価値を持つ個体が多数存在する場合には，集団内の多様性を保つためにリーパーキュー内の下位への移動量が減少し，評価値の良い個体もリーパー削除によって削除されやすいまま

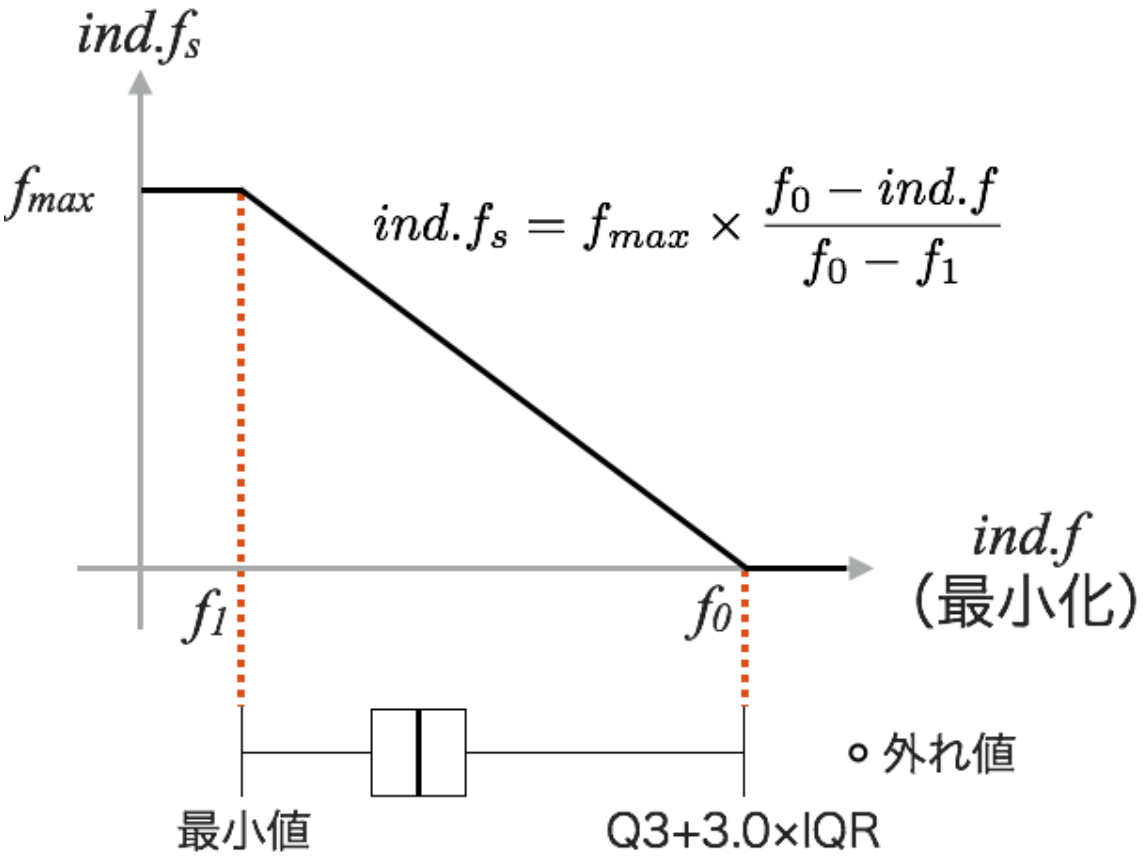


図 4.8　四分位数に基づく適合度スケーリングの概略図（最小化問題）

となる．逆に，最小の評価値と同等の評価値を持つ個体が少数の場合は，集団内に優良個体を保持するために下位への移動量を増加させ，優良個体の淘汰を抑制する．

4.4.3　TAEA$^+$ のアルゴリズム

QFS と P^{α}_{down} を加えた TAEA$^+$ のアルゴリズムを Algorithm 15 に示す．TAEA と同様に，各個体は（擬似的に）並列に評価され（1 行目），評価が完了するごとに ind の評価値 $ind.f$ を $list_f$ にサンプルする（2 行目）．$list_f$ は初期は空の状態から開始し，$list_f$ の要素が N_f に達した時に QFS の f_1 と f_0 を算出する（3 行目から 15 行目）．同時に P^{α}_{down} を $list_f$ に基づいて算出し（16 行目），$list_f$ を空の状態に戻す（17 行目）．以降，QFS で求まる適合度と適応的に調整した P^{α}_{down} を用いて，TAGP/TDTS と同様の処理を行い，個体を非同期に進化する．

Algorithm 15 適応型 TAEA（TAEA$^+$）のアルゴリズム

1: ind の適合度（$ind.f$）を計算
2: $list_f$ に $ind.f$ を追加
3: **if** $|list_f| = N_f$ **then**
4: $f_1 = min(f_1, list_f.Min)$
5: $tmp\_f_0 = min(f_0, list_f.Q3 + 3.0 \times list_f.IQR)$
6: **if** $|f_1 - tmp\_f_0| \neq 0$ **then**
7: $f_0 = tmp\_f_0$
8: **end if**
9: **if** $f_1 \geq list_f.Min$ **then**
10: $f_1\_not\_updated = 0$
11: **else**
12: $f_1\_not\_updated++$
13: **if** $f_1\_not\_updated \geq N_{update}$ **then**
14: $f_1 = list_f.Min$
15: **end if**
16: **end if**
17: P^{α}_{down} の算出（式 (4.4)）
18: $list_f = \emptyset$
19: **end if**
20: $ind.f_s = f_{max} \times \dfrac{f_0 - ind.f}{f_0 - f_1}$
21: $ind.f_{acc} \leftarrow ind.f_{acc} + ind.f_s$
22: （以降，Algorithm 13 の 3 行目から 20 行目と同様）

第 5 章

非同期リファレンス評価を用いる進化的アルゴリズム（Evolutionary Algorithm using Asynchronous Reference-based Evaluation：ARE-EA）

5.1 概要

本章では，TAEA の利点を引き継ぎつつ，その問題点を解消した相対評価に基づく非同期進化法として，第三の親選択とそのアーカイブを用いるリファレンス評価を用いる進化的アルゴリズム（Asynchronous Reference-based Evaluation EA：ARE-EA）を提案する本章では，はじめに ARE-EA の基本的な設計方針を示し，続いてそのアルゴリズムを述べる．

5.2 TAEA の問題点

前章で示した TAEA は絶対評価を用いて非同期に個体を進化可能であり，かつ評価の完了しない個体を待機時間を設けずにリーパーキューによって対処可能であり，特に（無限）ループを含むコンピュータプログラムの進化に有効である．一方，TAEA には下記のような問題がある：(1) 一般的な EA と異なり，TAEA では優良個体が親個体として選択される保証がなく，進化の妨げになる，優良個体の保持のためには，優良個体をそのまま集団内に複製するエリート保存戦略を導入しているが，エリート個体の選択，および保持

を保証するものではない．(2)TAEA では絶対評価に基づく閾値を用いて個体を選択，削除するため，適切に閾値を設定可能な問題に対してしか適用できない．一般に，問題に対する適切な設定は未知であるため，それらの問題には適用が困難である．

5.3 設計方針

5.3.1 アプローチ

上記の TAEA の問題を解決するために，本章で提案する ARE-EA では以下の 2 点を設計方針とする：(1) アーカイブによる優良個体の保持，(2) 評価時間が短く，適合度が低い個体の相対評価による淘汰．

(1) アーカイブによる優良個体の保持について，非同期進化においては一般に優良個体を集団内に保持することは困難であり，優良個体を常に親個体として選択できるとは限らない．そこで ARE-EA では，優良個体を保持するためのアーカイブを導入し，(a) 適合度の低い個体，(b) 評価時間の長い（もしくは評価が完了しない）個体を淘汰する．(a) 適合度の低い個体を淘汰する機構として，その個体がリファレンス個体（後述）と比較して劣る場合にその個体を削除の対象とする．また，(b) 評価時間が長い個体を淘汰する機構として，TAEA で用いられるリーパーキューを採用する．

次に (2) 評価時間が短く，適合度が低い個体の相対評価による淘汰について，非同期進化においては，高い適合度を持つ個体だけでなく，評価時間が短い個体も優先的に親個体として選択される機会を得る．しかし，評価時間の短い個体が必ずしも高い適合度を持つとは限らない．このような特徴から，非同期進化においては評価時間の短い個体が適合度にかかわらず親個体として選択される可能性が高くなってしまう．その結果，集団がこれらの子個体で満たされてしまい，局所解に陥りやすくなってしまう．このような状況を回避するために，ARE-EA において全個体は，既に評価が完了し，高い評価値を持つ第三の個体（リファレンス個体）と比較される．もし，評価の完了した個体がリファレンス個体よりも劣る場合には，その個体は親として選択されない．これにより，評価時間が短く適合度の低い個体を取り除きつつ，優良な個体を親として選択することが可能になる．ARE-EA では，リファレンス個体との相対的な比較を導入することにより，TAEA で必要であった絶対評価に基づく閾値設定が不要になる．具体的には，親選択と淘汰がリファレンス個体との比較によって実行されることで，非同期な進化を実現している．

5.3.2 TAEA との相違点

表 5.1 に ARE-EA と TAEA の差異をまとめる．ARE-EA と TAEA の主な違いは，(1)TAEA では進化のために累積適合度と選択のための閾値が必要であるが，ARE-EA で

表 5.1　TAEA と ARE-EA の相違点

	TAEA(/TDTS)	ARE-EA
親個体の選択	累積適合度に基づく閾値（+閾値を超えた個体との比較）	リファレンス個体との比較
優良個体の保持	優良個体のコピー	アーカイブ
低評価個体の削除	リーパーキュー内の順位に基づく削除	リファレンス個体との比較に基づく削除

はリファレンス個体との比較により親選択をするためこのような設定が必要なく，一般的な EA と同様に任意の適合度関数が使用可能である．これにより，ARE-EA は最適解が未知な問題に対しても適用可能になる．(2)TAEA は優良個体を保持するためにエリート保存戦略により優良個体のコピーを集団内に生成するが，リーパーキューの働きにより優良個体がからなずしも集団内に保持される保証がないが，ARE-EA ではアーカイブの機構により優良個体の保持が保証される．これにより，進化の際に優良個体を親として選択可能であり，さらにリファレンス個体との比較により適合度の低い個体を取り除くことが可能になる．(3)TAEA は低評価個体に対してリーパーキュー内の位置を変更することによって削除を促すが，ARE-EA ではリファレンス個体との比較により低評価であると判断されるとリーパーキュー内の順位に関係なく削除される．これにより，評価時間が短く低評価な個体が頻繁に親個体候補として選択されることなく削除可能になる．

5.4　ARE-EA

図 5.1 に ARE-EA の概略図を示す．ARE-EA は母集団であるリーパーキューと優良個体を保持するアーカイブを持ち，各個体はそのどちらかに配置される．アーカイブする個体の上限はアーカイブサイズ（as）で定義される．アーカイブは初期集団生成時は空であり，最初期に評価の完了した as 個体が初期アーカイブとなる．ARE-EA では，TAEA と同様に各個体は（擬似的に）並列に実行され，2 個体の評価が完了した段階で親個体を選択する（図 5.1 中 Step1）．親個体の選択は，評価が完了した 2 個体とアーカイブからランダムに選択されたリファレンス個体の 3 個体によるトーナメント選択によって選択される（図 5.1 中 Step2）．例えば，図 5.1 では，ind_1 と ind_2 の 2 個体の評価が完了した段階でアーカイブからランダムに選択されたリファレンス個体 ind_{ref} の 3 個体からトーナメント選択によって 2 個体の親個体 par_1 と par_2 を選択する．親選択後，2 体の親個体から交叉や突然変異などの遺伝的操作を適用して 2 体の子個体を生成し（図 5.1 中 Step3），リーパーキューの最下位に追加する（図 5.1 中 Step4）．その後，評価済みの 2 個体とリファレンス個体を比較し，リファレンス個体よりも優れている個体はアーカイブに保存す

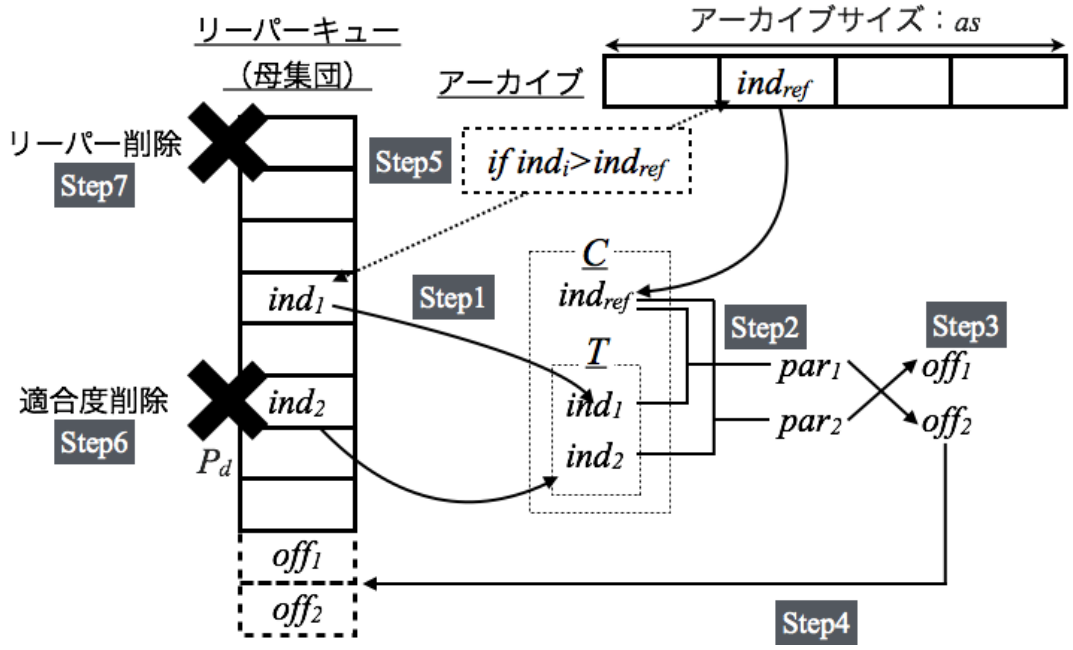


図 5.1　ARE-EA の概略図

る（図 5.1 中 Step5）．例えば，図 5.1 において ind_1 が ind_{ref} よりも優れている場合には，ind_1 がアーカイブに追加される．ARE-EA の特徴として，評価時間と適合度を考慮した 2 種類の削除機構がある．一つ目はリファレンス個体との比較に基づく削除で適合度削除と呼ぶ（図 5.1 中 Step6）．適合度削除では，評価済みの個体がリファレンス個体よりも劣る場合，一定の確率 P_d で削除される．以降，適合度削除に用いる確率 P_d を適合度削除確率と呼ぶ．例えば，図 5.1 において ind_2 が ind_{ref} より劣る場合，ind_2 は適合度削除確率 P_d の確率で削除される．適合度削除により，評価時間が短く適合度の低い個体の多くが削除される．適合度削除後に集団サイズが最大集団サイズを超過する場合，二つ目の削除機構であるリーパーキューに基づく削除を実行する．以降，この削除をリーパー削除と呼ぶ．（図 5.1 中 Step7）リーパー削除では，TAEA と同様にリーパーキューの上位の個体から順に削除する．例えば，図 5.1 において 2 個体が新たに生成され，適合度削除によって 1 個体のみが削除された場合，リーパー削除によってリーパーキューの上位 1 個体が追加で削除される．リーパー削除では，評価時間が長い個体や長い間生存し続けている個体が削除される．これらの削除において，適合度削除確率 P_d は適合度削除とリーパー削除とのバランスを決定するパラメータである．具体的には，P_d が高い場合には適合度削除の割合が高くなり，リーパー削除の割合が減少する．逆に，P_d が低い場合には適合度削除の割合が低くなり，リーパー削除の割合が増加する．リーパー削除の割合が高い場合，評価時間の長い個体が評価が完了する前に削除されるようになるため，P_d はリーパー削除がどれくらいの時間各個体の評価を待機するかを制御するパラメータとなる．

以下に ARE-EA の選択，子個体生成，適合度削除とアーカイブ生成，リーパー削除の各手順について詳細を示す．

5.4.1 選択

ARE-EA では，評価の完了した 2 個体（一時個体 T と呼ぶ）とアーカイブからランダムに選択されたリファレンス個体（ind_{ref} と表す）を合わせた 3 個体（$C = T \cup \{ind_{ref}\}$）の中から親個体を 2 個体選択する．

具体的には，3 個体からランダムに 2 個体を選択し，優れた個体を選択するトーナメント選択を 2 回実行し，2 個体を選択する．この時，親個体 2 個体の重複は許すものとする．

5.4.2 子個体生成

ARE-EA では，親個体として選択された個体をもとに一定の確率で交叉，突然変異，命令語の挿入・削除の遺伝的操作を実行し，新しい子個体を生成する．交叉は，親個体として選択された個体同士で実行され，交叉点は命令語単位で 2 個体独立に決定する．突然変異は任意の一命令をランダムに変化させる．命令語の挿入・削除は複製される個体の任意の位置に命令語を挿入，または削除する．生成された個体はリーパーキューの最下位（リーパー削除から最も遠い）に追加される．

5.4.3 適合度削除とアーカイブ生成

ARE-EA では，一時個体 T とリファレンス個体 ind_{ref} の比較に基づいて適合度削除，およびアーカイブの生成を行う．具体的な流れを Algorithm 16 に示す．ここで，一時個体の 2 個体をそれぞれ ind_{T1}，ind_{T2}（$ind_{T1} \leq ind_{T2}$），リファレンス個体を ind_{ref} とする．基本的な流れは，(1)ind_{ref} よりも劣る個体は適合度削除確率 P_d に基づいて削除，(2)ind_{ref} よりも優れている場合にはアーカイブに移動，(3) アーカイブサイズを超える場合には ind_{ref} をリーパーキューの最下位（リーパー削除から最も遠い）に移動となる．まず，ind_{ref} より劣る個体は適合度削除確率 P_d でリーパーキューから削除する（Algorithm 16 中 2，3 行目，および 5 行目）．次に，ind_{T2} が ind_{ref} より優れている場合，アーカイブ条件の判定を行う（6，12，17 行目）．具体的には，アーカイブ内に ind_{T2} と同一の個体が存在しない場合，あるいはアーカイブ内に ind_{ref} と同一の個体が存在する場合は ind_{T2} をアーカイブに移動する（7，13 行目）．これにより，アーカイブ内に ind_{T2} と同一の個体が存在し，かつ ind_{ref} と同一の個体がアーカイブ内に 1 個体のみの場合に ind_{ref} がアーカイブ内から失われることを防ぎ，アーカイブ内の個体の多様性を維持する．ここで，同一判定は扱う問題に応じて適合度，遺伝子の内容，表現形などによ

り決定する．ind_{T2} がアーカイブ条件を満たさない場合，ind_{T2} はリーパーキューの最下位に移動する（9，16 行目）．ind_{T1} が ind_{ref} より優れる場合は ind_{T1} もアーカイブの候補となる（11 行目以降）．もし ind_{T2} がアーカイブ条件を満たしてアーカイブに移動している場合には ind_{T1} はリーパーキューの最下位に移動し，ind_{T2} がアーカイブ条件を満たさない場合には ind_{T1} に同様にアーカイブ条件を適用し，アーカイブに移動するか，リーパーキューの最下位に移動するかを決定する（17 行目から 21 行目）．最後にアーカイブ内の個体数がアーカイブサイズ as を超過する（ind_{T1} もしくは ind_{T2} がアーカイブに追加された）場合，ind_{ref} をアーカイブ内からリーパーキューの最下位に移動する（24 行目から 26 行目）．

5.4.4 リーパー削除

ARE-EA では，集団（リーパーキュー）内の個体数が最大集団サイズを超過する場合，最大集団サイズ以下になるまでリーパーキューの上位数 %（例えば 30%）からランダムに個体を削除する．リーパーキューに基づく削除では長い間生き残った個体や適合度の低い個体が削除される．

5.4.5 アルゴリズム

ARE-EA のアルゴリズムを Algorithm 17 に示す．Algorithm 17 において，ind は評価の完了した個体を示し，$ind.f$ は個体 ind の適合度，MAX_POP は最大集団サイズを示す．ind_{ref} はリファレンス個体をあらわす．各個体は（擬似的に）並列に評価され（1 行目），2 個体（これらを一時個体 T と呼ぶ）の評価が完了した段階で親個体を選択する（3 行目）一時個体 T とアーカイブからランダムに選択されたリファレンス個体 ind_{ref} の 3 個体によるトーナメント選択によって親個体を 2 個体選択し（4 行目から 6 行目），交叉や突然変異などの遺伝的操作を適用して子個体を 2 個体生成する（7 行目）．生成された子個体はリーパーキューの最下位に追加される（8 行目）．その後，Algorithm 16 に基づいてアーカイブの生成と適合度削除を行う（9 行目）．適合度削除後に集団内の個体数が最大集団サイズ MAX_POP を超過する場合，リーパー削除によってリーパーキューの上位数 % の個体からランダムに削除し（10 行目から 12 行目），最後に一時個体を空にする（13 行目）以上の操作を繰り返すことで ARE-EA は非同期に個体を進化させる．

Algorithm 16 適合度削除，およびアーカイブ生成のアルゴリズム

1: **if** $ind_{T1} < ind_{T2} < ind_{ref}$ **then**
2: 　ind_{T1} を確率 P_d で削除
3: 　ind_{T2} を確率 P_d で削除
4: **else if** $ind_{T1} < ind_{ref} < ind_{T2}$ **then**
5: 　ind_{T1} を確率 P_d で削除
6: 　**if** ind_{T2} がアーカイブに存在しない $\vee$ ind_{ref} がアーカイブに存在する **then**
7: 　　ind_{T2} をアーカイブに移動
8: 　**else**
9: 　　ind_{T2} をリーパーキューの最下位に移動
10: 　**end if**
11: **else** $\{ind_{ref} < ind_{T1} < ind_{T2}\}$
12: 　**if** ind_{T2} がアーカイブに存在しない $\vee$ ind_{ref} がアーカイブに存在する **then**
13: 　　ind_{T2} をアーカイブに移動
14: 　　ind_{T1} をリーパーキューの最下位に移動
15: 　**else**
16: 　　ind_{T2} をリーパーキューの最下位に移動
17: 　　**if** ind_{T1} がアーカイブに存在しない $\vee$ ind_{ref} がアーカイブに存在する **then**
18: 　　　ind_{T1} をアーカイブに移動
19: 　　**else**
20: 　　　ind_{T1} をリーパーキューの最下位に移動
21: 　　**end if**
22: 　**end if**
23: **end if**
24: **if** アーカイブ内の個体数 $>$ アーカイブサイズ as **then**
25: 　ind_{ref} をリーパーキューの最下位に移動
26: **end if**

Algorithm 17 ARE-EA のアルゴリズム

1: ind を評価（$ind.f$）
2: $T \leftarrow ind$
3: **if** $|T| = 2$ **then**
4: 　ind_{ref}（リファレンス個体）をアーカイブからランダムに選択
5: 　$C \leftarrow T \cup \{ind_{ref}\}$
6: 　C から親個体を 2 個体を選択
7: 　遺伝的操作を行い子個体を 2 個体生成
8: 　子個体をリーパーキュー最下位に追加
9: 　適合度削除とアーカイブの生成（Algorithm 16）
10: 　**if** 集団内の個体数 >MAX_POP **then**
11: 　　リーパーキューの最上位個体から削除
12: 　**end if**
13: 　$T \leftarrow \emptyset$
14: **end if**

第 6 章

例題

本章では，実験で用いる例題について説明する．本研究では，(1)GP で一般的に用いられる関数同定問題（symbolic regression problem）における関数進化 [3]，(2) 実応用を想定した PIC マイコンで実行可能なアセンブリプログラム進化，(3) 宇宙機への適用を念頭に，宇宙環境においてプログラムのビット反転が発生する環境下でのアセンブリプログラム進化を扱う．以下，それぞれの例題について詳細を説明する．

6.1 関数同定問題

関数同定問題（symbolic regression problem）は GP の例題として広く一般に扱われており [38]，与えられた複数のデータ点を通る関数（プログラム）を獲得する問題である．具体的に関数同定問題では，図 6.1 に示すように，ある与えられた関数（図中実線）上のデータ点（図中円）が入出力対として与えられ，それらのデータ点をすべて通過する関数を生成することが目的となる．本研究では，各プログラムはそれぞれ $r0$ から $r7$ の 8 つの実数値レジスタ（変数）と $\{1, \cdots, 9\}$ の定数を使用可能とし，命令セットとして表 6.1 に示す 8 命令を用いる．ここで，表 6.1 において，dst，$src1$ には $r0$ から $r7$ のいずれかのレジスタが与えられ，$src2$ には $r0$ から $r7$ のいずれかのレジスタか定数が与えられる．例えば，ADD $r0$ $r1$ $r2$ は $r0 \leftarrow r1 + r2$ を表す．また，PDIV (%) は保護付き除算（protected division）を表し，分母が 0 の場合は予め定めた最大値を dst に代入し，それ以外の場合は通常の除算を行う．

例題として表 6.2 に示す 8 種類の関数を用いる [38]．入出力データ数はすべて 100 個とし，データ点は R1，R2，R3，R5，R6 では $-1 \leq x \leq 1$ の間で均等に，R7 では $0 \leq x \leq 2$ の間で均等に，R4 と R8 では $0.01 \leq x \leq 1$，$0 \leq y \leq 1$ の領域内で 10×10 のメッシュ状に均等にそれぞれ与える．実験は，5 から 25 のサイズでランダムに生成したプログラムで集団を満たした状態から開始する．

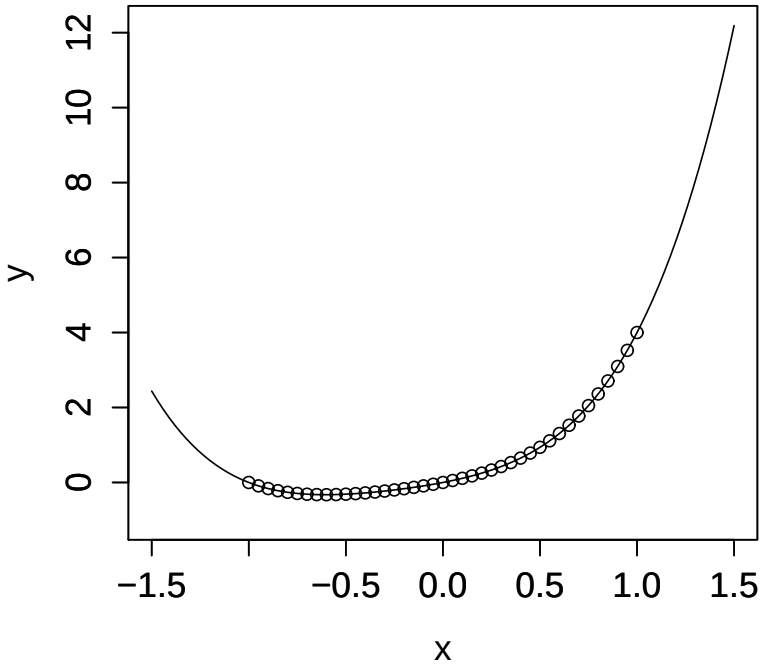


図 6.1　関数同定問題の概略図

表 6.1　関数同定問題で扱う命令セット

命令	操作
ADD dst $src1$ $src2$	$dst \leftarrow src1 + src2(const)$
SUB dst $src1$ $src2$	$dst \leftarrow src1 - src2(const)$
MUL dst $src1$ $src2$	$dst \leftarrow src1 \times src2(const)$
PDIV dst $src1$ $src2$	$dst \leftarrow src1\%src2(const)$
SIN dst $src1$	$dst \leftarrow sin(src1)$
COS dst $src1$	$dst \leftarrow cos(src1)$
LN dst $src1$	$dst \leftarrow ln(\|src1\|)$
EXP dst $src1$	$dst \leftarrow exp(src1)$

6.2　アセンブリプログラム進化問題

本例題では，アセンブリ命令で記述されたプログラムの進化を扱う．具体的には，Microchip Technology 社が製造している PIC10 [1] に組み込まれている 12 ビット，33

表 6.2　関数同定問題の例題

R1	$f(x) = x^4 + x^3 + x^2 + x$
R2	$f(x) = x^5 - 2x^3 + x$
R3	$f(x) = x^6 - 2x^4 + x^2$
R4	$f(x, y) = x^y$
R5	$f(x) = sin(x^2) \times cos(x) - 1$
R6	$f(x) = sin(x) + sin(x + x^2)$
R7	$f(x) = ln(x + 1) + ln(x^2 + 1)$
R8	$f(x, y) = sin(x) + sin(y^2)$

命令のアセンブリ言語で記述されたプログラムの実行ステップ数の最小化を目的とする．各プログラムは 16 個の汎用レジスタ（$r0$ から $r15$）と 1 個のワーキングレジスタ（W）と呼ばれる一時レジスタを持ち，それぞれ 32 ビットの符号なし整数値を持つ．このアセンブリ命令には表 6.3 に示すように単純な加減算，論理演算，ビット演算，および分岐命令が含まれるが，乗算命令が含まれないため，乗算を実現するためには加減算と論理演算のループが必要となる．ここで，表 6.3 において f は $r0$ から $r15$ の汎用レジスタ，d は 0 か 1 のビット，b は 5 ビットの符号なし整数値，k は 8 ビットの符号なし整数値で表される．1 番目から入力数分の汎用レジスタに入力値，それ以外の汎用レジスタに 0 を入力し，出力値は 0 番目の汎用レジスタの値を用いる．

例題として表 6.4 に示す 8 種類の計算を実行するプログラムを扱う．具体的には，4 種類の数値計算プログラムと 4 種類の論理演算プログラムを扱う．A1，A2，A3 の例題では，入力値は $\{0, \cdots, 15\}$ の符号なし整数値とする．A4 の例題では，入力値は $r1 = \{1, \cdots, 5\}$，$r2 = \{1, \cdots, 5\}$ の符号なし整数値の組み合わせ $(r1, r2)$ とする．B1 の 8bit-Parity は，図 6.2 に示すように $r1$ から $r8$ に 0 または 1 が入力され，1 の数が偶数の場合には $r0$ に 0，1 の数が奇数の場合には $r0$ に 1 を出力するプログラムである．例えば，図 6.2 では，入力された 1 の数が 4 個で偶数のため，$r0$ には 0 を出力する．B2 の 7bit-DigitalAdder は，図 6.3 に示すように $r1$ から $r7$ に 0 または 1 が入力され，$r1$ から $r3$ の三桁のビット列と $r4$ から $r6$ の三桁のビット列を $r7$ をキャリー（桁上げ）ビットとして加算した 4 ビットの結果を $r8$ から $r11$ に出力するプログラムである．例えば，図 6.3 では，$r1$ から $r7$ に “0111011” が入力されていることから，$011 + 101 + 1 = 1001$ を計算し，$r8$ から $r11$ にそれぞれ，“1”，“0”，“0”，“1” を出力する．B3 の 6bit-Multiplexer は，図 6.4 に示すように $r1$ から $r6$ に 0 または 1 が入力され，上位 2 ビット（$r1$ と $r2$）をアドレスビット，下位 4 ビット（$r3$ から $r6$）をデータビットと呼び，データビットは先頭から 0 番目，1 番目，2 番目，3 番目と数えられる．6bit-Multiplexer ではアドレスビットの 2 進数の値を 10 進数に変換し，その値が指し示すデータビットの値を $r0$ に出力

表 6.3　アセンブリプログラム進化で扱う PIC マイコン命令セット（[1] をもとに一部変更）

命令	操作
ADDWF f, d	$W \leftarrow W + f\ (d = 0)\ f \leftarrow W + f\ (d = 1)$
ANDWF f, d	$W \leftarrow W\ AND\ f\ (d = 0)\ f \leftarrow W\ AND\ f\ (d = 1)$
CLRF f	$f \leftarrow 0$
CLRW	$W \leftarrow 0$
COMF f, d	$W \leftarrow\sim f\ (d = 0)\ f \leftarrow\sim f\ (d = 1)$
DECF f, d	$W \leftarrow f - 1\ (d = 0)\ f \leftarrow f - 1\ (d = 1)$
DECFSZ f, d	DECF を実行後，結果が 0 なら次命令をスキップ
INCF f, d	$W \leftarrow f + 1\ (d = 0)\ f \leftarrow f + 1\ (d = 1)$
INCFSZ f, d	INCF を実行後，結果が 0 なら次命令をスキップ
IORWF f, d	$W \leftarrow W\ OR\ f\ (d = 0)\ f \leftarrow W\ OR\ f\ (d = 1)$
MOVF f, d	$W \leftarrow f\ (d = 0)\ f \leftarrow f\ (d = 1)$
MOVWF f	$f \leftarrow W$
RLF f, d	$W \leftarrow f << 1\ (d = 0)\ f \leftarrow f << 1\ (d = 1)$
RRF f, d	$W \leftarrow f >> 1\ (d = 0)\ f \leftarrow f >> 1\ (d = 1)$
SUBWF f, d	$W \leftarrow f - W\ (d = 0)\ f \leftarrow f - W\ (d = 1)$
SWAPF f, d	f の上位ビットと下位ビットを入れ替え
XORWF f, d	$W \leftarrow W\ XOR\ f\ (d = 0)\ f \leftarrow W\ XOR\ f\ (d = 1)$
BCF f, b	f の b ビット目を 0
BSF f, b	f の b ビット目を 1
BTFSC f, b	f の b ビット目が 0 の場合，次命令をスキップ
BTFSS f, b	f の b ビット目が 1 の場合，次命令をスキップ
LABEL k	k 番のラベル
GOTO k	k 番の *label* 命令に無条件ジャンプ
ANDLW k	$W \leftarrow W\ AND\ k$
IORLW k	$W \leftarrow W\ OR\ k$
MOVLW k	$W \leftarrow k$
XORLW k	$W \leftarrow W\ XOR\ k$

する．例えば，図 6.4 では，アドレスビットが “01”（= 1）であることから，データビットの 1 番目（$r4$）の値を $r0$ に出力する．B4 の 7bit-Majority は，図 6.5 に示すように $r1$ から $r7$ に 0 または 1 が入力され，入力された値の多い方を $r0$ に出力するプログラムである．例えば，図 6.5 では，入力値のうち，0 の数が 3 個，1 の数が 4 個であり 1 の入力が多いため，$r0$ に 1 を出力する．B1，B2，B3，B4 の例題では，各入力値は 0 または 1

表 6.4　アセンブリプログラム進化問題の例題

数値計算		入力数	出力数	データ数
A1	$f(x) = x^4 + x^3 + x^2 + x$	1	1	16
A2	$f(x) = x^5 - 2x^3 + x$	1	1	16
A3	$f(x) = x^6 - 2x^4 + x^2$	1	1	16
A4	$f(x, y) = x^y$	2	1	25
論理演算		**入力数**	**出力数**	**データ数**
B1	8bit-Parity	8	1	256
B2	7bit-DigitalAdder	7	4	128
B3	6bit-Multiplexer	6	1	64
B4	7bit-Majority	7	1	128

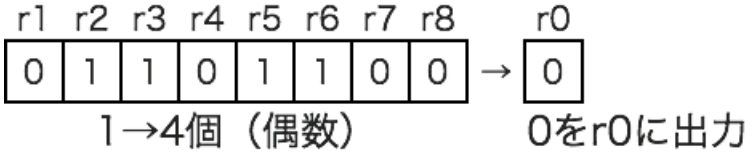


図 6.2　B1：8bit-Parity の概略図

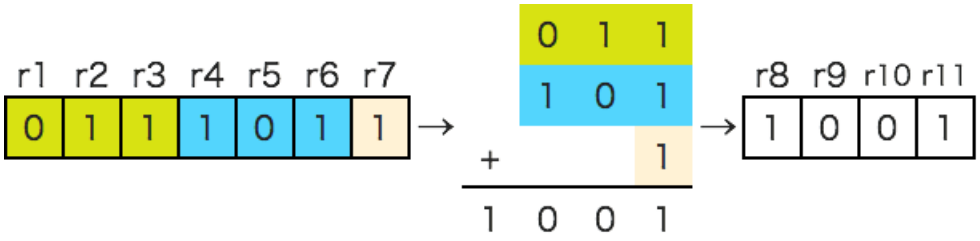


図 6.3　B2：7bit-DigitalAdder の概略図

とし，6 ビットから 8 ビットの全組み合わせを入力とする．

個体の 1 回の評価は，プログラムに対して入力値を与え，その入力に基づいてプログラムを最後まで実行し，その出力結果と入力値に対応する目標値を比較する操作をデータ数分の入出力対に対して行い完了する，

6.3　ビット反転下でのアセンブリプログラム進化問題

本例題では，実応用を考慮した例題として宇宙環境においてプログラムにビット反転が発生する環境下におけるプログラム進化を扱う．本節では，はじめに本例題の背景を説明し，続いて問題解決のアプローチとしてプログラムを進化させるオンボードコンピュータ

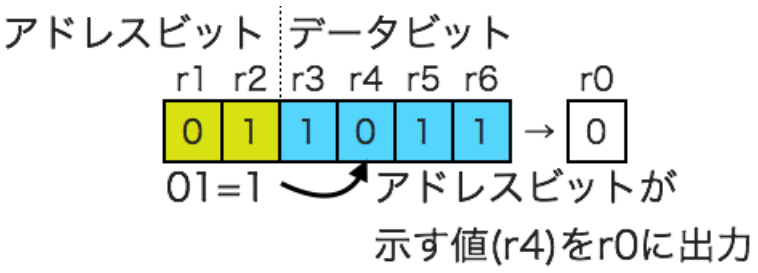


図 6.4　B3：6bit-Multiplexer の概略図

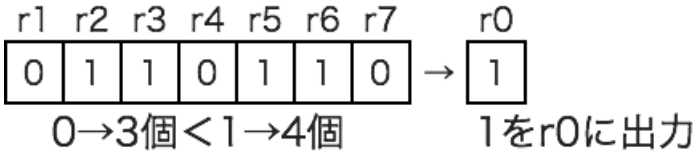


図 6.5　B4：7bit-Majority の概略図

について説明する．そして，最後に本例題で扱うプログラムを示す．

6.3.1　シングルイベントアップセット

宇宙空間では，地上よりも多くの宇宙線が絶えず飛び交っており，それがメモリや CPU など半導体デバイスに衝突すると，電荷の変化から 0/1 で記憶されている情報のビットが反転し，データの破壊や誤作動，最悪の場合はシステムの停止といった状況を引き起こす．これをシングルイベントアップセット（Single-Event Upset: SEU）[39] といい，入射するエネルギーが半導体デバイスに蓄えられるエネルギー量を超えたときに起こる．さらに，半導体デバイスの微細化に伴い，記憶されている情報が 1 ビット反転するシングルビットアップセット（Single-Bit Upset: SBU）だけでなく，複数のビットが反転するマルチビットマップセット（Multiple-Bit Upset: MBU）[40] の発生も確認されており，宇宙開発における半導体デバイスの宇宙線対策が重要になっている．実際，JAXA の民生部品・コンポーネント実証衛星「つばさ」(MDS-1) に搭載された民生半導体部品実験装置では，64MbitDRAM では SEU の 3 割程度が MBU と識別されている [41, 42]．

SEU の対策として，従来では (1) 金属のシールドを張る方法，(2) 論理回路の冗長化，(3) 配線幅の太い低性能 CPU の利用等のハードウェアによる対策がとられてきた．しかし，(1) に関しては衛星の重量が重くなり，制御のための燃料に莫大なコストが必要になり，(2) に関しては回路面積が余分に必要なため貴重なスペースが無駄になるだけでなく，多重化を制御する部分が故障するとシステムが機能しなくなるという問題がある．また，(3) に関しては計算能力が現在一般に利用されている CPU に比べ低く，高度な処理に限界がある．これに対し，チェックサム [43] などのソフトウェアによる対策も考えられる

表 6.5　ビット反転下でのアセンブリプログラム進化問題の例題

数値計算		入力数	出力数	データ数
A1	$f(x) = x^4 + x^3 + x^2 + x$	1	1	16
A2	$f(x) = x^5 - 2x^3 + x$	1	1	16
A3	$f(x) = x^6 - 2x^4 + x^2$	1	1	16
A4	$f(x, y) = x^y$	2	1	25
パリティチェック		入力数	出力数	データ数
E5	5bit-Parity	5	1	32
E6	6bit-Parity	6	1	64
E7	7bit-Parity	7	1	128
E8	8bit-Parity	8	1	256

が，MBU に弱く，完全には対処できない．

6.3.2　プログラムを進化させる宇宙機用オンボードコンピュータ

上記問題を解決するために，本研究ではコンピュータプログラムを生物，SEU によるビット反転をプログラムの突然変異とみなし，ビット反転をトリガーとしてプログラムを進化（効率化）させるオンボードコンピュータ（On-Board Computer: OBC）を提案する．この OBC では，従来防ぐべき宇宙線によるビット反転を逆に利用することによって SEU に耐性を持たせるだけではなく，より効率的なプログラムを生成する（進化させる）ことを可能にする．

6.3.3　例題で扱うプログラム

例題として表 6.5 に示す 8 種類の計算を実行するプログラムを扱い，プログラムサイズの最小化を目的とする．具体的には，4 種類の数値計算プログラムと 4 種類の偶数パリティプログラムを扱う．A1，A2，A3 の例題では，入力値は $\{0, \cdots, 15\}$ の符号なし整数値とする．A4 の例題では，入力値は $r1 = \{1, \cdots, 5\}$，$r2 = \{1, \cdots, 5\}$ の符号なし整数値の組み合わせ $(r1, r2)$ とする．E5，E6，E7，E8 の例題では，各入力値は 0 または 1 とし，5 ビットから 8 ビットの全組み合わせを入力とする．実験は，与えられた計算を実行可能なプログラムで集団を満たした状態から開始する．

第 7 章

実験 1：絶対評価を用いる非同期進化的アルゴリズム（TAEA）

7.1　概要

本章では, TAEA の有効性を検証するために, TAEA を遺伝的プログラミング (Genetic Programming：GP）に適用した TAGP を用いる実験を行う．まずはじめに 7.2 章では実応用を想定した PIC アセンブリプログラムの進化を用いる実験とその結果を示し，続いて 7.3 章で TAGP の宇宙機への適用を念頭においた SEU によるビット反転が起こる環境下で PIC アセンブリプログラムを進化させる実験とその結果を示す．最後に，7.4 章で GP の一般的なベンチマーク問題である関数同定問題を用いる実験とその結果を示す．

7.2　アセンブリプログラム進化

7.2.1　実験内容

TAGP と TD トーナメント選択（TDTS)，および TAGP$^+$ の有効性を検証するために，本実験では下記の実験を行う：

- **Case1:** TDTS を用いない TAGP (単に TAGP と表記) と TDTS を用いる TAGP（TAGP/TDTS と表記）の比較
- **Case2:** TDTS におけるトーナメントサイズ λ の比較
- **Case3:** TAGP/TDTS と適応的 TAGP（TAGP$^+$）の比較
- **Case4:** TAGP$^+$ と従来手法の比較

Case4 では，従来手法として同期 GP の steady-state GP（SSGP）[32] と $(\mu+\lambda)$-GP，非同期 GP の ASSGP[17] を扱う．Case1 と Case3，Case4 では，TDTS におけるトーナ

表 7.1　アセンブリプログラム進化の例題（6.2 章より再掲）

数値計算		入力数	出力数	データ数
A1	$f(x) = x^4 + x^3 + x^2 + x$	1	1	16
A2	$f(x) = x^5 - 2x^3 + x$	1	1	16
A3	$f(x) = x^6 - 2x^4 + x^2$	1	1	16
A4	$f(x, y) = x^y$	2	1	25
論理演算		**入力数**	**出力数**	**データ数**
B1	8bit-Parity	8	1	256
B2	7bit-DigitalAdder	7	4	128
B3	6bit-Multiplexer	6	1	64
B4	7bit-Majority	7	1	128

表 7.2　パラメータ設定

最大評価回数	10^6	交叉率	0.7
最大プログラムサイズ	256	突然変異率	0.1
母集団サイズ	100	命令挿入率	0.1
f_{max}	100	命令削除率	0.1

メントサイズは最小の $\lambda = 1$ に設定する（*i.e.*, ind_t と ind_{t-1} のみ比較）．Case2 では，$\lambda = \{1, 2, 3, 4, 5, 10, 20\}$ を比較する．例えば，$\lambda = 5$ の場合は，ind_t から ind_{t-5} の中から親個体が選択される．$\lambda = 1$ の場合は，Case1，Case3 と同様の設定である．Case3 では，TAGP/TDTS と TAGP$^+$ に加え，TAGP$^+$ において固定のリーパー制御パラメータ P_{down} を用いる STAGP$^+$（Static TAGP$^+$ の意）を比較し，適合度スケーリング，および適応的リーパー制御の有効性を検証する．

例題としては，表 7.1 に示す 4 種類の数値計算プログラムと 4 種類の論理演算プログラムの計 8 種類のアセンブリプログラムの進化を扱う．

7.2.2　評価基準と設定

各種法の共通のパラメータ設定を表 7.2 に示す．従来手法である SSGP，$(\mu + \lambda)$-GP，ASSGP において，無限ループにより評価が完了しない個体が存在する場合，途中で評価を打ち切る必要があるため，その上限を各例題において正常なプログラムが十分実行を完了可能な 50,000 命令とし，上限を超えた個体の適合度は $-\infty$ とする．TAGP(/TDTS)，および STAGP$^+$ において，リーパーキュー制御のパラメータ P_r（P_{down} および P_{up} の上限）は 0.99 に設定する．TAGP(/TDTS) において適合度関数は数値計算プログラムで

は式 (7.1)，論理演算プログラムでは式 (7.2) をそれぞれ用いる．

$$f_N = f_{max} - \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i^*| \tag{7.1}$$

$$f_E = f_{max} \times (1 - \frac{2}{n}\sum_{i=1}^{n}\delta(\hat{y}_i, y_i^*)), \delta(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}, \tag{7.2}$$

ここで，$\hat{y}_i$ は i 番目の入力に対するプログラムの出力結果，y_i^* は i 番目の入力に対する目標値，n はデータ数を表す．上記の適合度関数は，出力結果と目標値の誤差が 0 になる時に最大適合度 f_{max} となり，誤差が大きくなるほど適合度が小さくなるように設定している．また，(S)TAGP$^+$，$(\mu + \lambda)$-GP，ASSGP では，適合度関数は式 (7.3)，式 (7.4) をそれぞれ用いる．

$$f_N = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i^*| \tag{7.3}$$

$$f_E = \frac{2}{n}\sum_{i=1}^{n}\delta(\hat{y}_i, y_i^*), \delta(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}, \tag{7.4}$$

個体の優劣は，(1) 適合度，(2) 実行ステップ数，(3) プログラムサイズの順に比較し評価する．

実験は，与えられた目的を達成可能なプログラムで集団を満たした状態から開始する．各実験は 30 試行ずつ実施し，(1) 目的を達成可能な（*i.e.*, f_{max} の適合度を持つ）プログラムを維持できた割合，(2) 初期プログラムからの実行ステップ数の減少割合の 30 試行平均をもとに評価する．ここで，(1) 目的を達成可能なプログラムを維持できた割合は最大評価回数終了後に集団内に一個体でも目的達成可能なプログラムが存在する場合を維持できたと定義し，(2) 実行ステップ数の減少割合は集団中のプログラムの最小実行ステップ数をもとに評価する．なお，実行ステップ数は各例題についてすべての入出力対を計算し終わるまでに実行した命令数を意味する．また，本実験において (1) 目的を達成可能な（*i.e.*, f_{max} の適合度を持つ）プログラムを維持できた割合は全手法，全例題において 100% であったため，結果としては割愛する．

7.2.3 結果

Case1: TAGP と TAGP/TDTS の比較

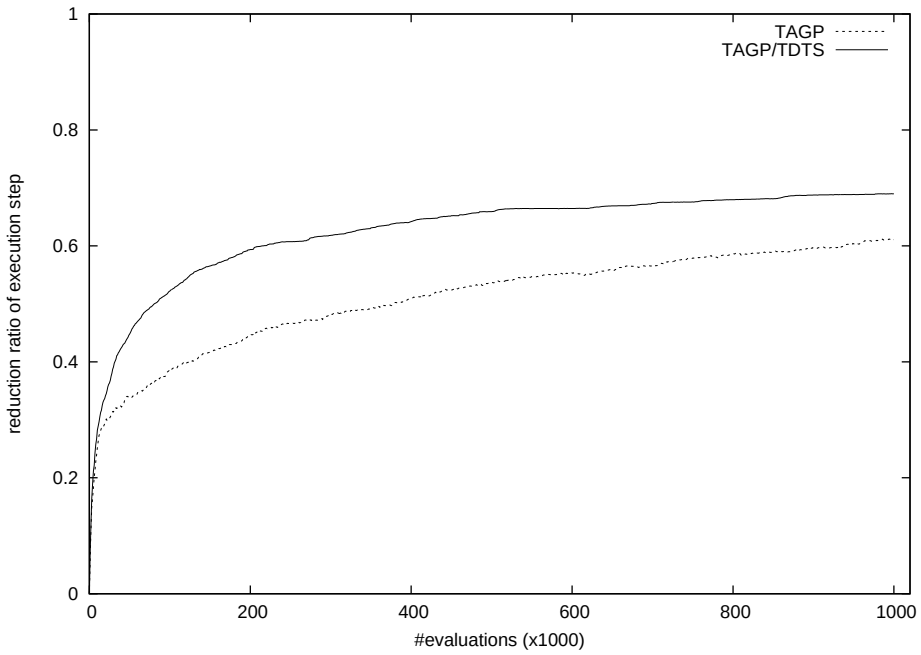
表 7.3 に TAGP と TAGP/TDTS の最大評価回数後の実行ステップ数減少割合の 30 試行平均を示す．各例題において最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．また，すべての試行，すべての例題において目的を達成可能なプログラムを維持できていることを確認している．これらの結果から，数値計算プログラムを扱う例題

表 7.3　最大評価回数後の実行ステップ数減少割合の 30 試行平均

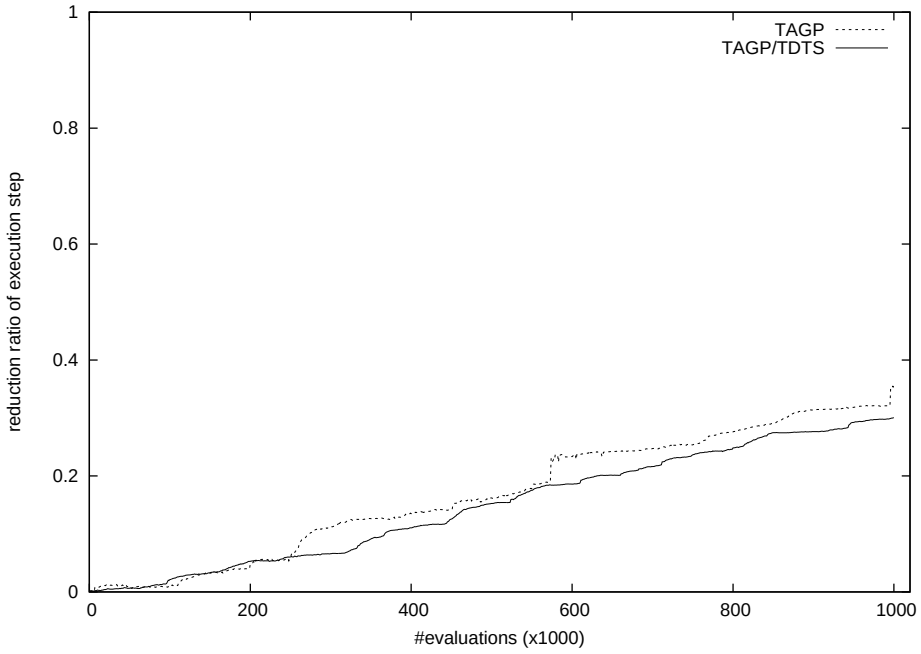
例題	TAGP	TAGP/TDTS	p 値
A1	61.1% (7.9%)	**71.2%** (6.6%)	< 0.01
A2	48.5% (5.4%)	**71.0%** (4.9%)	< 0.01
A3	49.7% (7.6%)	**70.5%** (6.6%)	< 0.01
A4	63.3% (13.3%)	**72.3%** (13.0%)	< 0.01
B1	**8.0%** (2.9%)	7.8% (3.0%)	1.25
B2	**4.8%** (2.6%)	4.2% (3.2%)	1.43
B3	56.2% (12.2%)	**57.5%** (10.5%)	0.65
B4	**35.8%** (24.2%)	30.1% (13.6%)	1.39

(A1∼A4) において TDTS を用いる TAGP (TAGP/TDTS) が TDTS を用いない TAGP を上回る性能を示し，論理演算プログラムを扱う例題（B1∼B4）では，TAGP/TDTS が TAGP と同等の性能を示すことが明らかになった．ノンパラメトリック検定の一つであるマン–ホイットニーの U 検定（Mann–Whitney U test）[44] を用いて検定を行った結果，数値計算プログラムにおいては有意水準 $\alpha = 0.05$ で有意差があることが確認された．実際，数値計算プログラムにおいては，TAGP/TDTS が TAGP に比べて 9%（A4）から 22.5%（A2）の実行ステップ数を削減できていることがわかる．TAGP/TDTS が数値計算プログラムの例題において TAGP を大きく上回る性能を示す理由は，プログラムが掛け算実行のためのループ構造を含むため実行ステップ数の大きなプログラムが生成されやすく，それを抑制するためにより高い選択圧が進化に必要なためである．一方，論理演算プログラムではループ構造が含まれず，実行ステップ数の差があまり生じないため，選択圧が低い TDTS を用いない TAGP でも十分に進化が可能なためである．

結果の詳細を示すため，図 7.1 に TAGP と TAGP/TDTS で大きな差が生じた A1 と同等の性能を示した B4 における実行ステップ数の減少割合の推移を示す．図 7.1 において，横軸は評価回数，縦軸は実行ステップ数の減少割合を表す．破線は TAGP の結果，実線は TAGP/TDTS の結果をそれぞれ示す．ここでは，A1 と B4 の結果のみ示しているが，他の A2∼A4，B1∼B3 についても同様の傾向を確認している．図 7.1(a) より，例題 A1 では進化の過程を通じて TAGP/TDTS の実行ステップ数減少割合が TAGP を常に上回っており，TDTS を用いることで進化が促進されていることがわかる．一方，図 7.1(b) より，例題 B4 では進化の過程を通じて TAGP と TAGP/TDTS がほぼ同じ実行ステップ数減少割合で推移している．これらの結果から，TAGP/TDTS が TAGP と比較して非同期なプログラム進化において，特にループ構造を含むプログラムの進化において，高い進化性能を示すことが明らかになった．



(a) 例題 A1：$f(x) = x^4 + x^3 + x^2 + x$



(b) 例題 B4：7bit-Majority

図 7.1　TAGP と TAGP/TDTS における実行ステップ数減少割合の推移：例題 A1 と例題 B4

Case2: TDTS におけるトーナメントサイズ λ の比較

表 7.4 に TDTS のトーナメントサイズ λ を変更した場合の最大評価回数後の実行ステップ数減少割合の 30 試行平均を示す．表 7.4 において，$\lambda = 1$ の結果は Case1 の TAGP/TDTS の結果と同一である．各例題において最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．また，すべての試行，すべての例題において目的を達成可能なプログラムを維持できていることを確認している．表 7.4 の結果から，A1，A3，B1 において $\lambda = 1$ が最も実行ステップ数を減少できており，他の例題においては，より大きなトーナメントサイズが優れた性能を示すことがわかる．しかし，マン–ホイットニーの U 検定の結果，これらのトーナメントサイズによる有意差は見られなかった．

表 7.4　TDTS のトーナメントサイズ λ を変更した場合の最大評価回数後の実行ステップ数減少割合の 30 試行平均

	トーナメントサイズ λ						
例題	1 (=Case1)	2	3	4	5	10	20
A1	**71.2%**	70.5%	70.0%	70.1%	69.1%	70.4%	70.9%
	(5.2%)	(5.7%)	(5.6%)	(5.5%)	(6.3%)	(5.5%)	(4.7%)
A2	71.0%	69.3%	70.8%	**72.5%**	70.8%	69.0%	70.5%
	(4.7%)	(4.7%)	(5.1%)	(4.2%)	(3.8%)	(4.3%)	(4.5%)
A3	**70.5%**	67.4%	68.1%	68.1%	69.5%	69.3%	66.2%
	(5.5%)	(5.2%)	(5.9%)	(6.3%)	(6.4%)	(4.6%)	(5.1%)
A4	72.3%	**75.4%**	69.9%	71.3%	71.6%	69.7%	72.4%
	(11.5%)	(5.4%)	(11.4%)	(11.3%)	(10.7%)	(12.7%)	(8.7%)
B1	**7.8%**	6.7%	6.3%	6.7%	7.5%	6.7%	7.2%
	(2.7%)	(3.0%)	(3.4%)	(3.5%)	(4.4%)	(4.2%)	(4.5%)
B2	4.2%	6.0%	6.0%	6.1%	**6.1%**	5.3%	5.1%
	(3.0%)	(3.4%)	(3.4%)	(4.1%)	(3.9%)	(3.5%)	(3.8%)
B3	57.5%	59.4%	**62.8%**	59.9%	57.1%	59.9%	59.9%
	(10.2%)	(12.2%)	(11.3%)	(10.0%)	(9.4%)	(9.1%)	(9.6%)
B4	30.1%	34.3%	32.8%	35.1%	33.2%	**35.8**%	33.8%
	(17.6%)	(15.4%)	(17.9%)	(16.8%)	(16.4%)	(15.6%)	(16.8%)

図 7.2 に例題 A1 と A2 の実行ステップ数削減割合の平均の推移を示す．図 7.2 において，横軸は評価回数，縦軸は実行ステップ数削減割合を表す．各線は，最も色の濃い線が $\lambda = 1$ の結果を示し，色が薄くなるに連れて大きなトーナメントサイズを示す．これらの

結果から，トーナメントサイズの大きな $\lambda = 10, 20$ の場合（図中最も色の薄い 2 本の線）は初期収束が早いものの，その後は実行ステップ数の減少量が少なくなり，最終的な実行ステップ数削減割合はトーナメントサイズの小さな $\lambda \leq 5$ が上回っていることがわかる．これは，トーナメントサイズが大きくなることによって，進化の初期においては優良個体をより優先的に選択して子個体を生成できるためステップ数を大幅に減少させることができるが，局所解に陥ってしまいその後抜け出せなくなるためである．トーナメントサイズが小さい場合は，比較する個体の数が少ないため親選択に多様性が生まれ，局所解に陥らずに探索することができる．

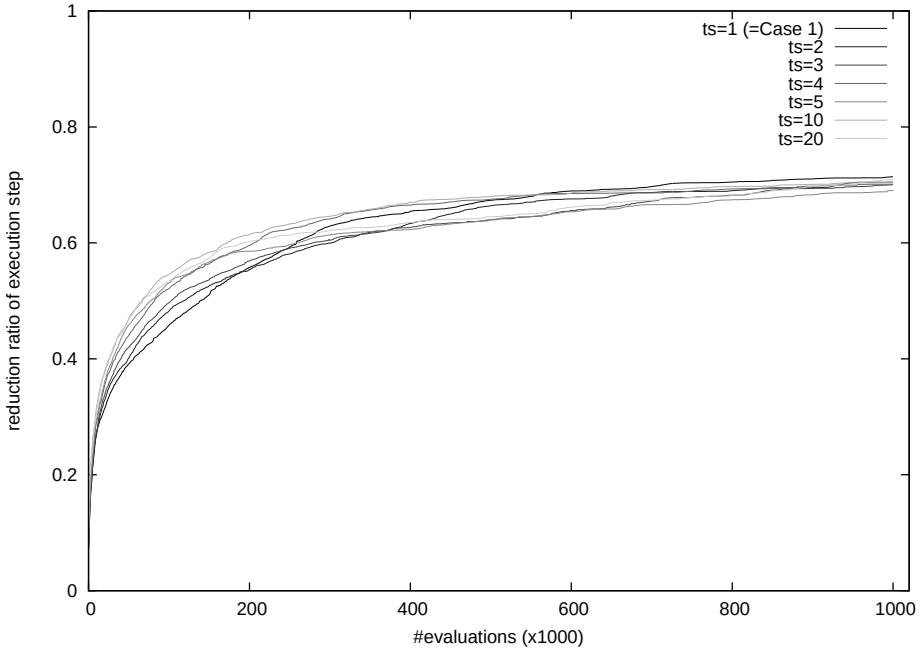
以上の結果から，トーナメントサイズは小さく設定することによって高い性能を実現することができ，$\lambda = 1$ に設定した場合でも十分な進化の性能を維持できることが明らかになった．

Case3: TAGP/TDTS と適応的 TAGP（TAGP$^+$）の比較

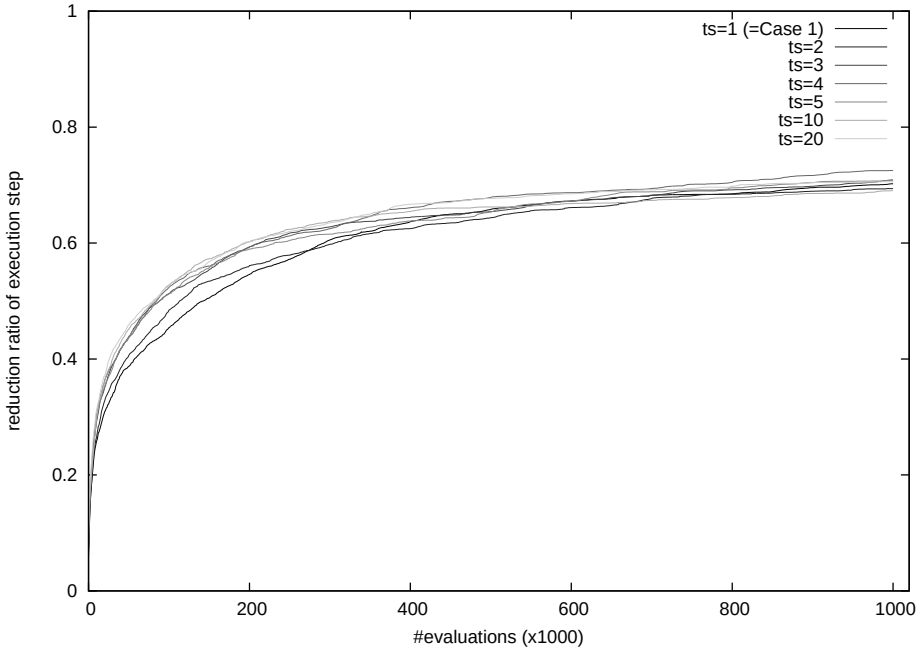
表 7.5 に TAGP/TDTS と固定のリーパー制御パラメータ P_{down} を用いる STAGP$^+$，適応的リーパー制御パラメータ P^{α}_{down} を用いる TAGP$^+$ の最大評価回数後の実行ステップ数減少割合の 30 試行平均を示す．表 7.5 において，p 値は TAGP/TDTS と TAGP$^+$ についてのマン–ホイットニーの U 検定の結果を示す．各例題において最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．また，すべての試行，すべての例題において目的を達成可能なプログラムを維持できていることを確認している．これらの結果から，非同期に得られる評価値に基づいて適合度関数を設定する (S)TAGP$^+$ が事前に適合度関数を設定する TAGP/TDTS と同等以上の進化性能を示すことが明らかになった．マン–ホイットニーの U 検定の結果，すべての例題において有意水準 5% で TAGP/TDTS と TAGP$^+$ に有意差が見られないことが明らかになった．このことから，TAGP$^+$ が適合度関数と閾値の事前の調整なしに最適化が可能であるといえる．

Case4: TAGP$^+$ と従来手法の比較

表 7.6 に従来手法である SSGP，$(\mu + \lambda)$-GP，ASSGP と TAGP$^+$ の最大評価回数後の実行ステップ数減少割合の 30 試行平均を示す．各例題において最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．表 7.6 において，p 値は TAGP$^+$ と従来手法で最も減少割合の高い手法（B3 は ASSGP，それ以外は SSGP）についてのマン–ホイットニーの U 検定の結果を示す．また，すべての試行，すべての例題において目的を達成可能なプログラムを維持できていることを確認している．これらの結果から，すべての例題において TAGP$^+$ が他の従来手法を上回る性能を示していることがわかる．マン–ホイットニーの U 検定の結果，A4 と B1 を除く例題において有意水準 5% で TAGP$^+$ と従来手法に有意差が見られることが明らかになった．



(a) 例題 A1：$f(x) = x4 + x^3 + x^2 + x$



(b) 例題 A2：$f(x, y) = x^5 - 2x^3 + x$

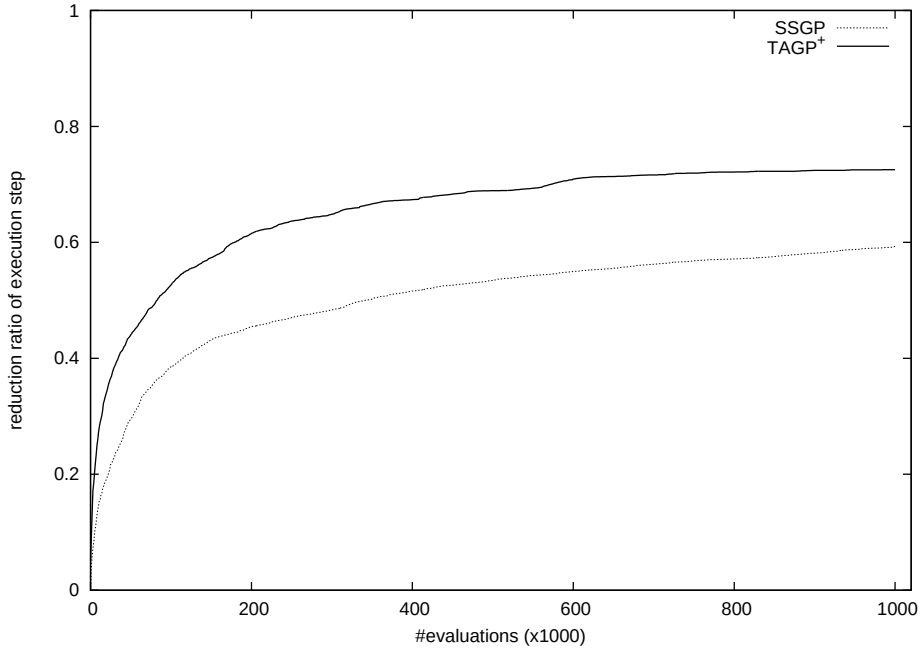
図 7.2　トーナメントサイズごとの実行ステップ数削減割合の平均の推移：例題 A1 と例題 A2

表 7.5　TAGP/TDTS と STAGP$^+$，TAGP$^+$ の最大評価回数後の実行ステップ数減少割合の 30 試行平均

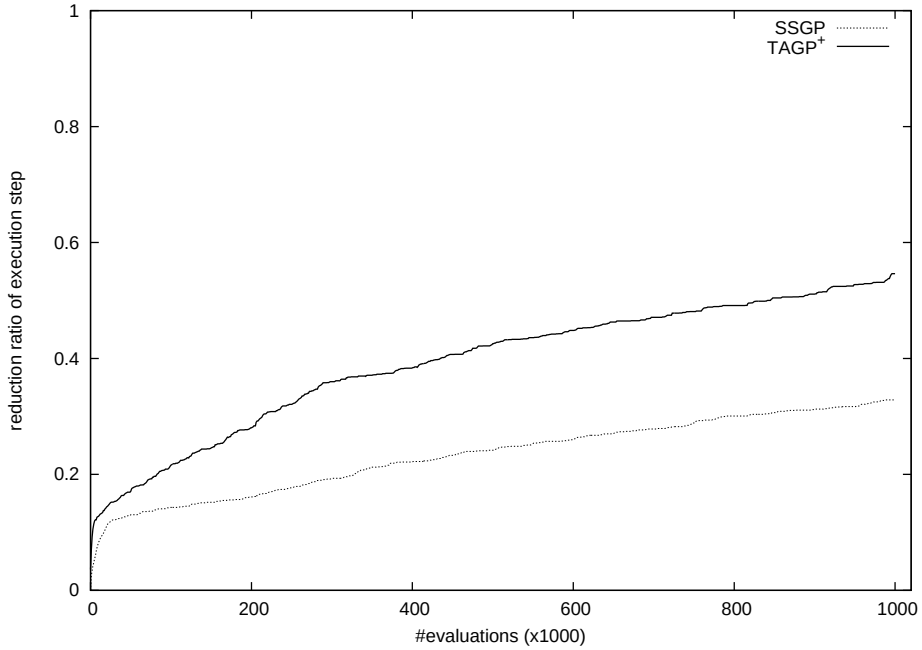
例題	TAGP/TDTS	STAGP$^+$	TAGP$^+$	p 値
A1	71.2% (6.6%)	**72.6%** (5.4%)	72.5% (5.2%)	0.43
A2	71.0% (4.9%)	70.9% (4.7%)	**72.9%** (4.8%)	0.43
A3	70.5% (6.6%)	71.2% (6.0%)	**71.4%** (5.5%)	0.06
A4	72.3% (13.0%)	73.4% (9.5%)	**76.4%** (7.9%)	0.36
B1	7.8% (3.0%)	7.3% (3.8%)	**8.3%** (3.7%)	0.72
B2	4.2% (3.2%)	**6.4%** (3.7%)	6.3% (7.5%)	0.52
B3	**57.5%** (13.6%)	54.4% (15.6%)	54.6% (18.9%)	0.37
B4	**30.1%** (13.6%)	24.2% (15.6%)	26.7% (18.9%)	0.76

結果の詳細を示すために，図 7.3 に SSGP と TAGP$^+$ で有意な差が見られた例題 A1 と B3 の実行ステップ数減少割合の推移，図 7.4 に有意な差が見られなかった例題 A4 と B1 の実行ステップ数減少割合の推移をそれぞれ示す．なお，ここでは減少割合の大きい SSGP と TAGP/TDTS のみを比較する．図 7.3，7.4 において，横軸は評価回数，縦軸は実行ステップ数減少割合を表す．破線は SSGP の結果，実線は TAGP$^+$ の結果をそれぞれ示す．

図 7.3 から，TAGP$^+$ の実行ステップ数減少割合が常に SSGP を上回っていることがわかる．この結果から，TAGP$^+$ は SSGP と比較して優れた進化性能を示していることがわかる．一方，図 7.4 から，A4 において TAGP$^+$ の実行ステップ数減少割合が進化の初期段階において SSGP を上回っていることがわかる．また，B1 においてはどちらの手法でも実行ステップ数の減少割合が非常に小さいことがわかる．これは，論理演算プログラムが数値計算プログラムとくらべて実行ステップ数を減少させる進化が困難であることを示している．このような進化が困難な問題においては，僅かな実行ステップ数の改善が重要と考えられることから TAGP$^+$ が高い進化性能を示しているといえる．

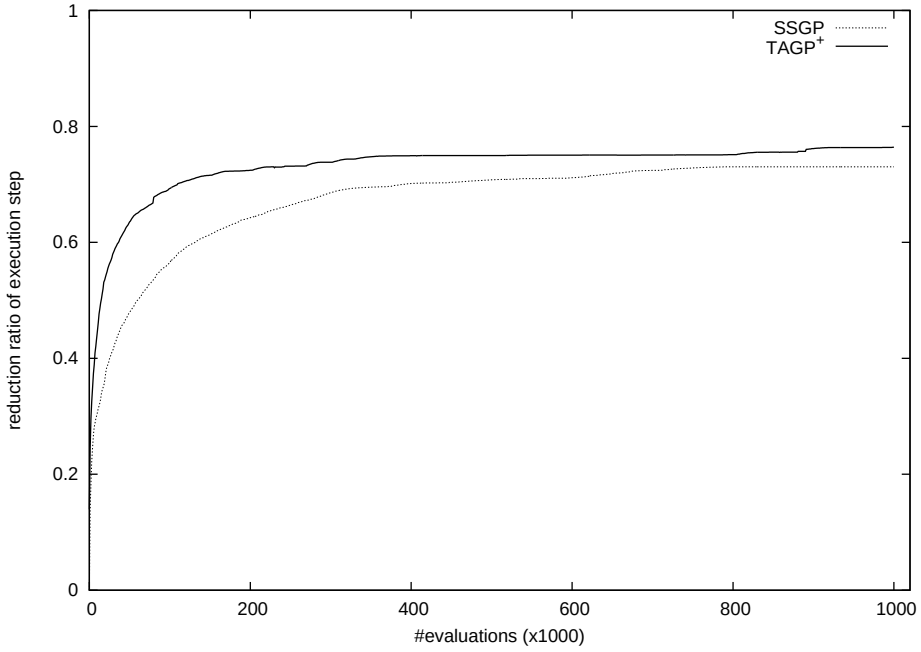


(a) 例題 A1：$f(x) = x4 + x^3 + x^2 + x$

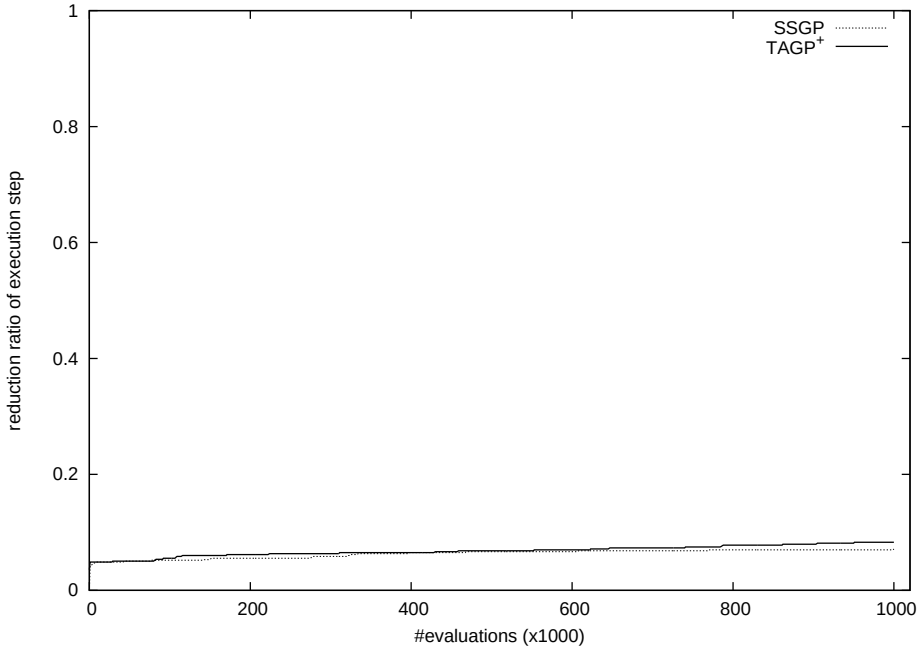


(b) 例題 B3：6bit-Multiplexer

図 7.3　SSGP と TAGP$^+$ における平均実行ステップ数の推移：例題 A1 と例題 B3



(a) 例題 A4：$f(x, y) = x^y$



(b) 例題 B1：8bit-Parity

図 7.4　SSGP と TAGP$^+$ における平均実行ステップ数の推移：例題 A4 と例題 B1

表 7.6　各手法の最大評価回数後の実行ステップ数減少割合の 30 試行平均

例題	SSGP	$(\mu+\lambda)$-GP	ASSGP	TAGP$^+$	p 値
A1	59.3% (4.6%)	5.7% (1.5%)	43.2% (8.4%)	**72.5%** (6.6%)	$< 10^{-3}$
A2	58.6% (4.4%)	5.8% (1.6%)	44.3% (6.5%)	**72.9%** (4.9%)	$< 10^{-3}$
A3	56.9% (4.3%)	6.1% (2.2%)	44.3% (7.8%)	**71.4%** (6.6%)	$< 10^{-3}$
A4	73.0% (9.6%)	21.3% (5.6%)	59.7% (12.7%)	**76.4**% (13.0%)	0.28
B1	7.2% (3.6%)	4.9% (0.0%)	5.8% (2.3%)	**8.3%** (3.0%)	0.69
B2	1.8% (2.3%)	0.4% (0.6%)	1.4% (2.3%)	**6.3%** (3.2%)	$< 10^{-3}$
B3	32.8% (8.4%)	4.2% (1.1%)	37.3% (10.4%)	**54.6%** (10.5%)	$< 10^{-3}$
B4	17.9% (16.7%)	0.6% (0.0%)	8.1% (10.8%)	**26.7%** (13.6%)	0.042

以上の結果から，TAGP$^+$ は数値計算プログラム，論理演算プログラムの両方において同期進化である SSGP，$(\mu+\lambda)$-GP，および従来の非同期 EA である ASSGP より高い進化性能を示すことが明らかになった．

7.2.4　考察 1：計算時間に対する性能の比較

非同期 EA は評価時間に差がある問題において，同期 EA と比較して短い時間で効率的な探索を実現できると考えられる．計算時間に対する TAGP$^+$ と同期 EA の SSGP，従来の非同期 EA の ASSGP の解探索性能の違いを明らかにする．ここでは，並列計算環境において各個体を並列に評価可能な環境を想定し，100 命令実行可能な時間を 1 単位時間とした時の経過単位時間における実行ステップ数の減少割合の 30 試行平均を比較する．表 7.7 に 10^7 単位時間経過後の各手法，各例題の実行ステップ数の平均減少割合を示す．各例題において最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．表 7.7 において，p 値は TAGP$^+$ と従来手法で最も減少割合の高い ASSGP についてのマン–ホイットニーの U 検定の結果を示す．これらの結果から，TAGP$^+$ が 10^7 単位時間で Case4 で示した 10^6 評価回数と同等の実行ステップ減少割合を達成しているのに対し，

SSGP は減少割合が著しく低下していることがわかる．このことから，同期 EA である SSGP と比べて，TAGP$^+$ は非同期な進化によって評価時間の無駄なく効率的な進化を実現できていることが明らかになった．また，従来の非同期 EA である ASSGP と比べても，TAGP$^+$ がすべての例題で高い減少割合を示していることがわかる．マン–ホイットニーの U 検定の結果，B1 を除くすべての例題で有意水準 5% で TAGP$^+$ と ASSGP に有意差が見られることが明らかになった．

以上の結果から，TAGP$^+$ は実行時間の観点から特に同期 EA と従来の非同期 EA を上回る高い進化性能を示すことが明らかになった．

表 7.7　各手法の 10^7 単位時間経過後の実行ステップ数減少割合の 30 試行平均

例題	SSGP	ASSGP	TAGP$^+$	p 値
A1	16.4% (2.5%)	37.7% (5.8%)	**68.5%** (7.7%)	$< 10^{-3}$
A2	4.0% (16.7%)	35.8% (6.5%)	**68.0%** (6.4%)	$< 10^{-3}$
A3	16.5% (3.9%)	35.8% (5.9%)	**65.6%** (6.2%)	$< 10^{-3}$
A4	38.0% (2.5%)	52.9% (10.6%)	**73.4%** (9.8%)	$< 10^{-3}$
B1	5.5% (2.1%)	7.3% (2.7%)	**7.5%** (3.3%)	0.93
B2	0.0% (0.2%)	0.5% (1.1%)	**4.1%** (6.7%)	$< 10^{-3}$
B3	16.3% (4.1%)	34.3% (12.0%)	**68.3%** (10.0%)	$< 10^{-3}$
B4	1.5% (2.7%)	2.1% (4.9%)	**5.0%** (9.7%)	0.048

7.2.5　考察 2：生成されたプログラム

TAGP/TDTS と TAGP$^+$，SSGP の性能の違いを明らかにするために，各手法で例題 A1 と例題 B3 において最終的に獲得されたプログラムを示す．プログラムは各手法 30 試行の中で最も実行ステップ数の短いプログラムを示している．

A1：$f(x) = x^4 + x^3 + x^2 + x$

図 7.5，7.6，7.7 に例題 A1 で最終的に獲得されたプログラムの一部を示す．図 7.5，7.6，7.7 に示すプログラムはいずれも SSGP，TAGP/TDTS，TAGP$^+$ で 30 試行の中で最も実行ステップ数の短いプログラムである．図 7.5，7.6 では例題 A1 において $f(x) = x^4 + x^3 + x^2 + x$（$x$ は 0 から 15 の 4 ビットが入力される）の x^2 を計算する部分，図 7.7 は x^4 を計算する部分をそれぞれ示している．各図のプログラムにおいて，R1，R2，R4，R5，R6，R7 はそれぞれ汎用レジスタを表しており，W はワーキングレジスタを表している．入力値（x）は R1 レジスタに入力され，x^2 の計算結果は一時的に R2 レジスタに，x^4 の計算結果は一時的に R4 レジスタにそれぞれ記憶される（最終的には出力用の R0 レジスタに加算される）．“< –” は数値の代入を表し，“>> 1” は右シフトを表す．

```
 1: MOVF    R1 0 // W <- R1
 2: MOVWF   R5 1 // R5 <- W
 3: MOVWF   R7 1 // R7 <- W
 4: MOVLW   32   // W <- 32
 5: MOVWF   R6 1 // R6 <- W
 6: MOVF    R5 0 // W <- R5
 7: BTFSC   R7 0 // if R7[0] == 0 then skip
 8: ADDWF   R2 1 // R2 <- R2 + W
 9: RRF     R2 1 // R2 <- R2 >> 1
10: RRF     R7 1 // R7 <- R7 >> 1
11: DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
12: BTFSC   R7 0 // if R7[0] == 0 then skip
13: ADDWF   R2 1 // R2 <- R2 + W
14: RRF     R2 1 // R2 <- R2 >> 1
15: RRF     R7 1 // R7 <- R7 >> 1
16: DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
17: BTFSC   R7 0 // if R7[0] == 0 then skip
18: ADDWF   R2 1 // R2 <- R2 + W
19: RRF     R2 1 // R2 <- R2 >> 1
20: RRF     R7 1 // R7 <- R7 >> 1
21: DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
22: BTFSC   R7 0 // if R7[0] == 0 then skip
23: ADDWF   R2 1 // R2 <- R2 + W
24: RRF     R2 1 // R2 <- R2 >> 1
25: DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
26: NOP     16 0 // label(0)
27: RRF     R2 1 // R2 <- R2 >> 1
28: DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
29: RRF     R2 1 // R2 <- R2 >> 1
30: DECFSZ  R6 1 // R6 <- R6 - 1 and if R6 = 0 then skip
31: GOTO    16   // goto label(0)
```

図 7.5　SSGP によって得られたプログラムの一部（例題 A1）

SSGP によって得られたプログラム（図 7.5）では，1 行目から 6 行目でレジスタを初期化し，続く 7 行目から 11 行目で入力値の 1 ビット目（最下位ビット），12 行目から 16

```
 1: MOVF    R1 0 // W <- R1
 2: MOVWF   R5 1 // R5 <- W
 3: MOVWF   R7 1 // R7 <- W
 4: BTFSC   R7 0 // if R7[0] == 0 then skip
 5: ADDWF   R2 1 // R2 <- R2 + W
 6: RRF     R2 1 // R2 <- R2 >> 1
 7: RRF     R7 1 // R7 <- R7 >> 1
 8: BTFSC   R7 0 // if R7[0] == 0 then skip
 9: ADDWF   R2 1 // R2 <- R2 + W
10: RRF     R2 1 // R2 <- R2 >> 1
11: RRF     R7 1 // R7 <- R7 >> 1
12: BTFSC   R7 0 // if R7[0] == 0 then skip
13: ADDWF   R2 1 // R2 <- R2 + W
14: RRF     R2 1 // R2 <- R2 >> 1
15: RRF     R7 1 // R7 <- R7 >> 1
16: BTFSC   R7 0 // if R7[0] == 0 then skip
17: ADDWF   R2 1 // R2 <- R2 + W
18: RRF     R2 1 // R2 <- R2 >> 1
19: RRF     R2 1 // R2 <- R2 >> 1
20: RRF     R2 1 // R2 <- R2 >> 1
...[RRF     R2 1] x 17 ...
37: RRF     R2 1 // R2 <- R2 >> 1
38: RRF     R2 1 // R2 <- R2 >> 1
39: RRF     R2 1 // R2 <- R2 >> 1
40: RRF     R2 1 // R2 <- R2 >> 1
41: RRF     R2 1 // R2 <- R2 >> 1
42: RRF     R2 1 // R2 <- R2 >> 1
43: RRF     R2 1 // R2 <- R2 >> 1
44: RRF     R2 1 // R2 <- R2 >> 1
45: RRF     R2 1 // R2 <- R2 >> 1
46: RRF     R2 1 // R2 <- R2 >> 1
```

図 7.6　TAGP/TDTS によって得られたプログラムの一部（例題 A1）

行目で 2 ビット目，17 行目から 21 行目で 3 ビット目，22 行目から 25 行目で 4 ビット目に対して 1 ビットずつ右シフトをしながら計算を実行する（入力は最大 4 ビット）．その後，出力結果を計算するために 26 行目から 31 行目のループによって 4 ビット右シフトされたレジスタ値をもとに戻す操作を実行し，計算結果が R2 レジスタに記憶される．一方，TAGP/TDTS によって得られたプログラム（図 7.6）では，1 行目から 3 行目でレジスタを初期化し，続く 4 行目から 7 行目で入力値の 1 ビット目（最下位ビット），8 行目から 11 行目で 2 ビット目，12 行目から 15 行目で 3 ビット目，16 行目から 18 行目で 4 ビット目に対して 1 ビットずつ右シフトをしながら計算を実行するその後，SSGP で得られたプログラムと同様に 4 ビット右シフトされたレジスタ値をもとに戻すために，図 7.6 では 19 行目から 46 行目でループを用いずにレジスタ値をもとに戻す操作を実行し，計算結果が R2 レジスタに記憶される．最後に，TAGP$^+$ によって得られてプログラム（図 7.7）では，TAGP/TDTS で得られたプログラムと同様にループを展開して R4 レジスタの値をもとに戻しているが，TAGP/TDTS で得られたプログラムが 20 回の右シフトを

```
 1: BTFSC    R7 0// if R7[0] == 0 then skip
 2: ADDWF    R4 1// R4 <- R4 + W
 3: RRF      R4 1// R4 <- R4 >> 1
 4: RRF      R7 1// R7 <- R7 >> 1
-------------------------------------------------
... BTFSC    R7 0// if R7[0] == 0 then skip
... ADDWF    R4 1// R4 <- R4 + W
... RRF      R4 1// R4 <- R4 >> 1
... RRF      R7 1// R7 <- R7 >> 1
    x 7
-------------------------------------------------
33: BTFSC    R7 0// if R7[0] == 0 then skip
34: ADDWF    R4 1// R4 <- R4 + W
35: RRF      R4 1// R4 <- R4 >> 1
36: RRF      R7 1// R7 <- R7 >> 1
37: BTFSC    R7 0// if R7[0] == 0 then skip
38: ADDWF    R4 1// R4 <- R4 + W
39: RRF      R4 1// R4 <- R4 >> 1
40: RRF      R7 1// R7 <- R7 >> 1
41: BTFSC    R7 0// if R7[0] == 0 then skip
42: ADDWF    R4 1// R4 <- R4 + W
43: RRF      R4 1// R4 <- R4 >> 1
44: RRF      R4 1// R4 <- R4 >> 1
45: RRF      R4 1// R4 <- R4 >> 1
46: RRF      R4 1// R4 <- R4 >> 1
47: RRF      R4 1// R4 <- R4 >> 1
48: SWAPF    R4 1// R4の上位16ビットと下位16ビットを交換
```

図 7.7　TAGP$^+$ によって得られたプログラムの一部（例題 A1）

実行してレジスタ値を戻しているのに対し，TAGP$^+$ で得られたプログラムはこの操作をより簡略化している．具体的には，図 7.7 では右シフトを 4 回だけ実行し，合計で 16 回の右シフトが実行された段階でレジスタ値の上位 16 ビットと下位 16 ビットを入れ替える SWAPF 命令を実行することによってレジスタ値をもとに戻している．これにより，TAGP/TDTS で得られたプログラムが 20 回の右シフトを実行していた処理を 4 回の右シフトと 1 回の SWAPF 命令で簡略化することにより，さらなる実行ステップ数の削減を実現している．

SSGP と TAGP/TDTS，TAGP$^+$ によって得られたプログラムの大きな違いは下記の 2 点である：(1)SSGP，TAGP/TDTS，TAGP$^+$ で得られたプログラムのサイズ（命令数）はそれぞれ 179 命令，177 命令，163 命令であり，TAGP/TDTS，TAGP$^+$ で得られたプログラムがループを展開しているにもかかわらず SSGP よりも簡潔なプログラムとなっており，さらに実行ステップ数も SSGP が 3563 ステップ，TAGP/TDTS が 2599 ステップ，TAGP$^+$ が 2375 ステップと TAGP/TDTS，TAGP$^+$ で得られたプログラム

の方が短くなっている．(2)SSGP で得られたプログラムはループを用いるためのループカウンタが必要となり，R6 レジスタを用いてカウントをしており（5，11，16，21，25，28，30 行目），かつループ操作のための GOTO 命令と行き先を示す LABEL 命令が必要となる（26 行目と 31 行目）．一方，TAGP/TDTS，TAGP$^+$ で得られたプログラムはループを用いない（ループを展開している）ためこれらの命令が必要なく，実行ステップ数を大幅に削減している．

最後に，SSGP と TAGP/TDTS，TAGP$^+$ で得られたプログラムはいずれも進化の際に与えられる入力値に特化したプログラムとなっている．具体的には，進化時に評価に用いる入力値が最大 4 ビットであるため，その目的を達成するためには TAGP/TDTS で得られたプログラムのようにその 4 ビット以外を無視して実行することが最も効率的である．しかし，これにより評価に用いる入力値以外（例えば $x = 16$）が与えられた場合には正しく答えられない．そのため，今回の実験で得られたプログラムでは評価時に与えられる入力値以外の入力に対する汎用性は失われていることに注意が必要である．

B3：6bit-Multiplexer

図 7.8，7.9，7.10，7.11 に例題 B3 の初期プログラム，および最終的に獲得されたプログラムを示す．図 7.9，7.10，7.11 に示すプログラムはいずれも SSGP，TAGP/TDTS，および TAGP$^+$ で 30 試行の中で最も実行ステップ数の短いプログラムである．図 7.8，7.9，7.10，7.11 のプログラムにおいて，R0 から R7 はそれぞれ汎用レジスタを表しており，W はワーキングレジスタを表している．6 つの 0 または 1 の入力値は R1 から R6 レジスタに入力され，R0 レジスタに結果が出力される．“$<-$” は数値の代入を表す．

まず，初期プログラム（図 7.8）では，出力結果を計算するために $r0 \leftarrow (\neg r1 \wedge \neg r2 \wedge r3) \vee (\neg r1 \wedge r2 \wedge r4) \vee (r1 \wedge \neg r2 \wedge r5) \vee (r1 \wedge r2 \wedge r6)$ という論理演算を実行している．具体的には，図中で色分けされた各処理において，それぞれ $(\neg r1 \wedge \neg r2 \wedge r3)$（2 行目から 12 行目），$(\neg r1 \wedge r2 \wedge r4)$（13 行目から 22 行目），$(r1 \wedge \neg r2 \wedge r5)$（23 行目から 32 行目），$(r1 \wedge r2 \wedge r6)$（33 行目から 41 行目）を算出し，$IORWF$ 命令を用いてそれぞれの実行結果の論理和を $r0$ に代入している．これに対し，SSGP で得られたプログラム（図 7.9）では，最初の 2 つの論理演算（$(\neg r1 \wedge \neg r2 \wedge r3)$ と $(\neg r1 \wedge r2 \wedge r4)$）は進化によって簡略化されているものの初期プログラムと同様であり，入力値の 1 ビット目（$r1$）の値が 0 の場合の結果はこの処理によって完了する（2 行目から 15 行目），しかし，プログラムの残りの部分（16 行目から 22 行目）では論理演算ではなく条件分岐によって出力結果を決定している．具体的には，入力値の 1 ビット目（$r1$）が 1 の場合には，入力値の 2 ビット目（$r2$）の値に応じて 0 であれば $r5$ の値，1 であれば $r6$ の値が出力値となるため，SSGP の進化後のプログラムでは 19 行目の $BTFSC$ 命令を用いて条件分岐し，出力結果を決定する．本実験で使用している PIC アセンブリプログラムでは，汎用レジスタ間

```
 1: CLRF     R0 1     // R0 <- 0
 2: MOVLW    1        // W  <- 1
 3: MOVWF    R7 1     // R7 <- W
 4: MOVLW    1        // W  <- 1
 5: XORWF    R1 0     // W  <- R1 XOR W
 6: ANDWF    R7 1     // R7 <- R7 AND W
 7: MOVLW    1        // W  <- 1
 8: XORWF    R2 0     // W  <- R2 XOR W
 9: ANDWF    R7 1     // R7 <- R7 AND W
10: MOVF     R3 0     // W  <- R3
11: ANDWF    R7 0     // W  <- R7 AND W
12: IORWF    R0 1     // R0 <- R0 OR W
13: MOVLW    1        // W  <- 1
14: MOVWF    R7 1     // R7 <- W
15: MOVLW    1        // W  <- 1
16: XORWF    R1 0     // W  <- R1 XOR W
17: ANDWF    R7 1     // R7 <- R7 AND W
18: MOVF     R2 0     // W  <- R2
19: ANDWF    R7 1     // R7 <- R7 AND W
20: MOVF     R4 0     // W  <- R4
21: ANDWF    R7 0     // W  <- R7 AND W
22: IORWF    R0 1     // R0 <- R0 OR W
23: MOVLW    1        // W  <- 1
24: MOVWF    R7 1     // R7 <- W
25: MOVF     R1 0     // W  <- R1
26: ANDWF    R7 1     // R7 <- R7 AND W
27: MOVLW    1        // W  <- 1
28: XORWF    R2 0     // W  <- R2 XOR W
29: ANDWF    R7 1     // R7 <- R7 AND W
30: MOVF     R5 0     // W  <- R5
31: ANDWF    R7 0     // W  <- R7 AND W
32: IORWF    R0 1     // R0 <- R0 OR W
33: MOVLW    1        // W  <- 1
34: MOVWF    R7 1     // R7 <- W
35: MOVF     R1 0     // W  <- R1
36: ANDWF    R7 1     // R7 <- R7 AND W
37: MOVF     R2 0     // W  <- R2
38: ANDWF    R7 1     // R7 <- R7 AND W
39: MOVF     R6 0     // W  <- R6
40: ANDWF    R7 0     // W  <- R7 AND W
41: IORWF    R0 1     // R0 <- R0 OR W
```

図 7.8　例題 B3 の初期プログラム

でのデータの受け渡しはできず一度ワーキングレジスタ（W）に代入してから目的のレジスタに代入する必要がある．そのため，汎用レジスタ間での論理演算（例えば，$r1 \wedge r2$）を計算する場合にも，一度値をワーキングレジスタに代入してから計算をする（例えば，$W \leftarrow r0$ ののち $W \leftarrow r1 \wedge W(= r0)$ を実行）必要があるため実行ステップ数が増加する．これに対し，条件分岐はワーキングレジスタを介さず直接汎用レジスタの値を検査して実行することができるため論理演算よりも実行ステップ数を削減することができる．一方，TAGP/TDTS によって得られたプログラム（図 7.10）では，SSGP で得られたプロ

```
 1: MOVLW    1    // W  <- 1
 2: XORWF    R1 0 // W  <- R1 OR W
 3: MOVWF    R7 1 // R7 <- W
 4: XORWF    R2 0 // W  <- R2 XOR W
 5: ANDWF    R7 1 // R7 <- R7 AND W
 6: MOVF     R3 0 // W  <- R3
 7: ANDWF    R7 0 // W  <- R7 AND W
 8: IORWF    R0 1 // R0 <- R0 OR W
 9: DECFSZ   R1 0 // W  <- R1 - 1
                     and if R1 == 0 then skip
10: MOVWF    R7 1 // R7 <- W
11: MOVF     R2 0 // W  <- R2
12: ANDWF    R7 1 // R7 <- R7 AND W
13: MOVF     R4 0 // W  <- R4
14: ANDWF    R7 0 // W  <- R7 AND W
15: IORWF    R0 1 // R0 <- R0 OR W
16: XORWF    R1 0 // W  <- R1 XOR W
17: MOVWF    R7 1 // R7 <- W
18: MOVF     R5 0 // W  <- R5
19: BTFSC    R2 0 // if R2[0] == 0 then skip
20: MOVF     R6 0 // W  <- R6
21: ANDWF    R7 0 // W  <- R7 AND W
22: IORWF    R0 1 // R0 <- R0 OR W
```

図 7.9　SSGP によって得られたプログラム（例題 B3）

```
 1: MOVF     R3 0 // W  <- R3
 2: DECFSZ   R2 1 // R2 <- R2 - 1
                     and if R2 == 0 then skip
 3: BTFSC    R3 5 // if R3[5] == 0 then skip
                     (always true)
 4: MOVF     R4 0 // W  <- R4
 5: BTFSS    R1 0 // if R1[0] == 1 then skip
 6: MOVWF    R0 1 // R0 <- W
 7: MOVF     R5 0 // W  <- R5
 8: BTFSS    R2 5 // if R2[5] == 1 then skip
                     (input R2 == 0)
 9: MOVF     R6 0 // W  <- R6
10: BTFSS    R1 0 // if R1[0] == 1 then skip
11: BTFSC    R8 1 // if R8[1] == 0 then skip
                     (always true)
12: IORWF    R0 1 // R0 <- R0 OR W
```

図 7.10　TAGP/TDTS によって得られたプログラム（例題 B3）

グラムから更に最初の 2 つの論理演算も条件分岐で実行することによって実行ステップ数を削減している．具体的には，1 行目から 4 行目で入力値の 1 ビット目（$r1$）の値に応じて 0 であれば $r3$，1 であれば $r4$ の値をワーキングレジスタ（W）に代入し，5 行目と 6 行目の条件分岐で入力値の 2 ビット目（$r2$）の値を検査し，0 であれば出力結果である $r0$ レジスタにワーキングレジスタの値を代入している．次に，7 行目から 9 行目で入力値の 2 ビット目（$r2$）の値に応じて 0 であれば $r5$，1 であれば $r6$ の値をワーキングレジ

```
 1: SUBWF    R3 0 // W <- R3 - W (W = 0)
 2: BTFSC    R2 0 // if R2[0] == 0 then skip
 3: MOVF     R4 0 // W <- R4
 4: BTFSS    R1 0 // if R1[0] == 1 then skip
 5: IORWF    R0 1 // R0 <- R0 OR W
 6: MOVF     R5 0 // W <- R5
 7: BTFSC    R2 0 // if R2[0] == 0 then skip
 8: MOVF     R6 0 // W <- R6
 9: BTFSC    R1 0 // if R1[0] == 0 then skip
10: IORWF    R0 1 // R0 <- R0 OR W
```

図 7.11　TAGP$^+$ によって得られたプログラム（例題 B3）

```
1:W←R3
2:if R2==1 then skip
3:if TRUE  then skip
4:W←R4
```

(a) 2 つの連続した条件分岐命令を用いる条件分岐（TAGP/TDTS で得られたプログラム（図 7.10）中 1 行目から 4 行目）

```
1:W←R3
2:if R2==0 then skip
3:W←R4
```

(b) 1 つの条件分岐命令を用いる条件分岐（ARE-GP で得られたプログラム（図 7.11）中 1 行目から 3 行目）

図 7.12　TAGP$^+$ で得られたプログラムにおける条件分岐の簡略化

スタ（W）に代入し，10 行目から 12 行目の条件分岐で入力値の 1 ビット目（$r1$）の値を検査し，1 であれば出力結果である $r0$ レジスタにワーキングレジスタの値を代入している．このように，TAGP/TDTS で得られたプログラムは論理演算の処理をすべて条件分岐に置き換えることによってさらなる実行ステップ数の削減が実現されている．さらに，TAGP$^+$ によって得られたプログラム（図 7.11）では，TAGP/TDTS で得られたプログラムと同様に条件分岐のみで計算を完了することで実行ステップ数を削減するとともに，無駄な命令が削減されていることがわかる．具体的には，TAGP/TDTS のプログラム中の 1 行目から 4 行目では図 7.12(a) に示すように $r1$ のレジスタの値に応じてワーキングレジスタ（W）に値を代入しているが，このとき 2 つの連続した条件分岐命令，「$r1$ の値が 1 の時に 1 命令スキップする命令」と「無条件に 1 命令スキップする命令」を用いて条件分岐の処理を実行している．しかし，この 2 つの連続した条件分岐命令は図 7.12 に示すような 1 つの「$r1$ の値が 0 の時に 1 命令スキップする命令」と等価である．そして，TAGP$^+$ ではこの等価な命令での置換えたプログラムが実際に獲得されており，これにより実行ステップ数が更に削減されている．

SSGP と TAGP/TDTS, TAGP$^+$ によって得られたプログラムの大きな違いは，SSGP で得られたプログラムは一部のみが条件分岐になっており，実行ステップ数が多く必要な論理演算の処理を残しているのに対し，TAGP/TDTS，TAGP$^+$ で得られたプログラム

表 7.8　ビット反転下でのアセンブリプログラム進化の例題（6.3 章より再掲）

数値計算		入力数	出力数	データ数
A1	$f(x) = x^4 + x^3 + x^2 + x$	1	1	16
A2	$f(x) = x^5 - 2x^3 + x$	1	1	16
A3	$f(x) = x^6 - 2x^4 + x^2$	1	1	16
A4	$f(x, y) = x^y$	2	1	25
パリティチェック		入力数	出力数	データ数
E5	5bit-Parity	5	1	32
E6	6bit-Parity	6	1	64
E7	7bit-Parity	7	1	128
E8	8bit-Parity	8	1	256

はすべてを条件分岐で実行することによって実行ステップ数が削減できているという点である．さらに TAGP$^+$ では，TAGP/TDTS で得られたプログラム中の冗長な条件分岐を簡略化することによって実行ステップ数を削減できている．

7.3　ビット反転環境下でのアセンブリプログラム進化

7.3.1　実験内容

本実験では，SEU によるビット反転が生じる環境において TAGP が PIC アセンブリプログラムを進化させられるかどうかを検証する．

例題としては，表 7.8 に示す 4 種類の数値計算プログラムと 4 種類の偶数パリティプログラムの計 8 種類のアセンブリプログラムの進化を扱う．

7.3.2　評価基準と設定

TAGP のパラメータを表 7.9 に示す．なお，本実験ではトーナメント選択 $\lambda = 1$ の TDTS を用いる TAGP/TDTS を扱い，リーパーキュー制御のパラメータ P_r（P_{down} および P_{up} の上限）は 0.99 に設定する．また，実応用ではメモリ容量が限られることが想定されるため，本実験では母集団サイズを比較的小さい 20 としている．また，単位時間は 100 命令の実行に要する時間と定義する．SEU は単位時間あたりに $\{10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$ の確率でプログラム，またはレジスタのうち 1 ビットが反転するものとする．適合度関数

表 7.9　パラメータ設定

最大単位時間	2.0×10^8	交叉率	0.7
最大プログラムサイズ	256	突然変異率	0.1
母集団サイズ	20	命令挿入率	0.1
f_{max}	100	命令削除率	0.1

は数値計算プログラムでは式 (7.5)，論理演算プログラムでは式 (7.6) をそれぞれ用いる．

$$f_A = f_{max} - \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i^*| \tag{7.5}$$

$$f_E = f_{max}(1 - \frac{2}{n}\sum_{i=1}^{n} \delta(\hat{y}_i, y_i^*)), \delta(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}, \tag{7.6}$$

ここで，$\hat{y}_i$ は i 番目の入力に対するプログラムの出力結果，y_i^* は i 番目の入力に対する目標値，n はデータ数を表す．個体の優劣は，(1) 適合度，(2) プログラムサイズ，(3) 実行ステップ数の順に比較し評価する．

実験は，与えられた目的を達成可能なプログラムで集団を満たした状態から開始する．各実験は 30 試行ずつ実施し，(1) 目的を達成可能な（*i.e.*, f_{max} の適合度を持つ）プログラムを維持できた割合，(2) 初期プログラムからのプログラムサイズの減少割合の 30 試行平均，(3) 初期プログラムからの実行ステップ数の減少割合の 30 試行平均をもとに評価する．ここで，(2) プログラムサイズの減少割合と (3) 実行ステップ数の減少割合は集団中のプログラムの最小実行ステップ数をもとに評価し，目的を達成可能なプログラムを維持できた試行における平均とする．また，実行ステップ数は各例題についてすべての入出力対を計算し終わるまでに実行した命令数を意味する．

7.3.3　結果

表 7.10 に (1) 目的を達成可能なプログラムを維持できた割合，表 7.11 に (2) プログラムサイズの減少割合の 30 試行平均，表 7.12 に (3) 実行ステップ数の減少割合の 30 試行平均をそれぞれ示す．なお，表 7.11，7.12 において，負値は減少率，正値は増加率を表す．まず，表 7.10 の結果から，TAGP はビット反転率が最も高い 10^{-3} の場合を除く全てのケースで目的を達成可能なプログラムを維持できていることがわかる．また，ビット反転率が最も高い 10^{-3} の場合も A2，A4 を除くケースでは目的を達成可能なプログラムを維持できており，A2，A4 においても維持できていないのは 30 試行中 1 試行ずつであった．今回実験で設定したビット反転率はいずれも実宇宙環境で観測されるビット反転率よりも高く設定しているため，TAGP を適用した OBC は SEU によるビット反転に十分耐性があると言える．次に，表 7.11 の結果から，A4 を除くすべての例題，ビット反転

表 7.10　(1) 目的を達成可能なプログラムを維持できた割合

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
A1	1.0	1.0	1.0	1.0
A2	0.97	1.0	1.0	1.0
A3	1.0	1.0	1.0	1.0
A4	0.97	1.0	1.0	1.0
E5	1.0	1.0	1.0	1.0
E6	1.0	1.0	1.0	1.0
E7	1.0	1.0	1.0	1.0
E8	1.0	1.0	1.0	1.0

表 7.11　(2) プログラムサイズの減少割合の 30 試行平均

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
A1	−27.3%	−26.6%	−27.2%	−26.8%
A2	−18.4%	−19.5%	−21.9%	−22.3%
A3	−24.6%	−25.2%	−23.9%	−15.5%
A4	32.4%	−5.9%	−9.9%	20.4%
E5	−21.8%	−23.6%	−22.4%	−19.6%
E6	−10.2%	−14.7%	−12.0%	−12.9%
E7	−7.9%	−7.4%	−7.0%	−5.8%
E8	−5.4%	−4.4%	−5.1%	−5.1%

表 7.12　(3) 実行ステップ数の減少割合の 30 試行平均

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
A1	13.4%	17.8%	12.4%	18.6%
A2	16.2%	13.0%	9.6%	19.7%
A3	10.5%	17.4%	14.5%	13.8%
A4	−0.3%	4.8%	7.4%	4.8%
E5	−3.6%	−4.1%	−8.4%	−4.8%
E6	16.6%	13.3%	15.2%	11.0%
E7	20.0%	17.0%	17.5%	18.3%
E8	16.0%	32.9%	31.8%	25.9%

率においてプログラムサイズを減少できていることがわかる．A4 においてプログラムサイズが増加している理由は，A4 の例題では $f(x,y) = x^y$ を計算するために初期プログラムには二重ループが含まれ，A1 から A3 までの一重ループよりもループの展開が生じやすいためである．一方，表 7.12 の結果から，実行ステップ数はほとんどの例題，ビット反転率で増加していることがわかる．これは，本実験においてプログラムサイズの最小化を目的としたためである．以上の結果から，TAGP は進化の過程で SEU によるビット反転が生じる環境においても目的を達成可能なプログラムを維持可能なだけでなく，初期に与えたプログラムよりも小さなプログラムを生成可能であることが明らかになった．

7.3.4 考察

進化によって実際に獲得されたプログラムの例を図 7.13，7.14 に示す．図 7.13 は，例題 A1（$f(x) = x^4 + x^3 + x^2 + x$）における初期プログラムと進化後のプログラム，図 7.14 は，例題 E5（5bit-Parity）における初期プログラムと進化後のプログラムをそれぞれ表す．

まず，例題 A1 において，初期プログラムでは x^2，x^3，x^4 の計算をそれぞれ 3 つのループ（初期プログラム 9 行目から 17 行目，27 行目から 35 行目，45 行目から 53 行目）で実行し R2，R3，R4 レジスタに記憶している．そして，最後に ADDWF 命令を 4 回実行し，R1（入力値 x），R2，R3，R4 の値を順に出力用のレジスタである R0 に加算している．それに対し，進化後のプログラムでは，交叉によって初期プログラムにおける 1 番目のループ（初期プログラム 9 行目から 17 行目）が 3 番目のループの箇所（進化後プログラム 31 行目から 37 行目）にコピーされている．これにより，x^2 の値が R2 レジスタに記憶された状態からさらに 3 番目のループで R2 レジスタに x^4 の計算結果が順次加算されている．そして，最終的に ADDWF 命令によって R0 レジスタに出力結果を計算する際に R2 レジスタに $x^4 + x^2$ が計算されているため，ADDWF 命令を 1 回分削減することができており，その分のプログラムサイズを削減されている．

次に，例題 E5 において，初期プログラムでは R6 レジスタに入力値である R1 から R5 までのレジスタ値を加算し（初期プログラム 2 行目から 11 行目），最後に R6 レジスタの 0 ビット目の値に基づいて出力値を決定している．この時，PIC マイコンの命令では汎用レジスタから汎用レジスタへの代入命令が存在しないため，W レジスタを一度経由してから計算をしている．それに対し，進化後のプログラムでは，R6 レジスタを用いずに入力値の一つである R5 レジスタを直接用いて計算しているため，R6 レジスタの初期化分の命令が削減されている（進化後プログラム 1 行目から 8 行目）．さらに，初期プログラムでは R1 レジスタから R5 レジスタまでに 1 が入力された個数を計算するために加算命令（ADDWF 命令）を使用しているため，最終的な出力を計算するために BTFSC（レジスタ値の指定したビットが 0 の場合，次の命令をスキップ）によって合計値の偶奇判定を

初期プログラム

```
 1: CLRF     R2 1      // R2 <- R1 x R1
 2: MOVF     R1 0      // W <- R1
 3: MOVWF    R5 1      // R5 <- W
 4: MOVWF    R6 1      // R6 <- W
 5: MOVF     R6 0      // W <- R6
 6: MOVWF    R7 1      // R7 <- W
 7: MOVLW    32        // W <- 32
 8: MOVWF    R6 1      // R6 <- 8
 9: LABEL0
10: MOVF     R5 0      // W <- R5
11: BCF      STATUS 0
12: BTFSC    R7 0      // if R7[0] == 0 then skip
13: ADDWF    R2 1      // R2 <- R2 + W
14: RRF      R2 1      // R2 <- R2 >> 1
15: RRF      R7 1      // R7 <- R7 >> 1
16: DECFSZ   R6 1      // R6 <- R6 - 1
                          and if R6 == 0 then skip
17: GOTO     LABEL0
18: CLRF     R3 1      // R3 <- R2 x R1
19: MOVF     R1 0      // W <- R1
20: MOVWF    R5 1      // R5 <- W
21: MOVF     R2 0      // W <- R2
22: MOVWF    R6 1      // R6 <- W
23: MOVF     R6 0      // W <- R6
24: MOVWF    R7 1      // R7 <- W
25: MOVLW    32        // W <- 32
26: MOVWF    R6 1      // R6 <- 8
27: LABEL1
28: MOVF     R5 0      // W <- R5
29: BCF      STATUS 0
30: BTFSC    R7 0      // if R7[0] == 0 then skip
31: ADDWF    R3 1      // R3 <- R3 + W
32: RRF      R3 1      // R3 <- R3 >> 1
33: RRF      R7 1      // R7 <- R7 >> 1
34: DECFSZ   R6 1      // R6 <- R6 - 1
                          and if R6 == 0 then skip
35: GOTO     LABEL1
36: CLRF     R4 1      // R4 <- R3 x R1
37: MOVF     R1 0      // W <- R1
38: MOVWF    R5 1      // R5 <- W
39: MOVF     R3 0      // W <- R3
40: MOVWF    R6 1      // R6 <- W
41: MOVF     R6 0      // W <- R6
42: MOVWF    R7 1      // R7 <- W
43: MOVLW    32        // W <- 32
44: MOVWF    R6 1      // R6 <- 8
45: LABEL2
46: MOVF     R5 0      // W <- R5
47: BCF      STATUS 0
48: BTFSC    R7 0      // if R7[0] == 0 then skip
49: ADDWF    R4 1      // R4 <- R4 + W
50: RRF      R4 1      // R4 <- R4 >> 1
51: RRF      R7 1      // R7 <- R7 >> 1
52: DECFSZ   R6 1      // R6 <- R6 - 1
                          and if R6 == 0 then skip
53: GOTO     LABEL2
54: CLRF     R0 1      // R0 <- 0
55: MOVF     R1 0      // W <- R1
56: ADDWF    R0 1      // R0 <- R0 + W
57: MOVF     R2 0      // W <- R2
58: ADDWF    R0 1      // R0 <- R0 + W
59: MOVF     R3 0      // W <- R3
60: ADDWF    R0 1      // R0 <- R0 + W
61: MOVF     R4 0      // W <- R4
62: ADDWF    R0 1      // R0 <- R0 + W
```

図 7.13　例題 A1（$f(x) = x^4 + x^3 + x^2 + x$）における初期プログラムと進化後のプログラム（1/2）

進化後プログラム

```
 1: MOVF     R1 0     // W <- R1
 2: MOVWF    R5 1     // R5 <- W
 3: IORWF    R7 1     // R7 <- R7 OR W (R7 <- W)
 4: MOVLW    32       // W <- 32
 5: IORWF    R6 1     // R6 <- R6 OR W (R6 <- W)
 6: LABEL0
 7: MOVF     R5 0     // W <- R5
 8: BTFSC    R7 0     // if R7[0] == 0 then skip
 9: ADDWF    R2 1     // R2 <- R2 + W
10: RRF      R2 1     // R2 <- R2 >> 1
11: RRF      R7 1     // R7 <- R7 >> 1
12: DECFSZ   R6 1     // R6 <- R6 - 1
                         and if R6 == 0 then skip
13: GOTO     LABEL0
14: MOVF     R2 0     // W <- R2
15: MOVWF    R7 1     // R7 <- W
16: MOVLW    32       // W <- 32
17: MOVWF    R6 1     // R6 <- W
18: LABEL1
19: MOVF     R5 0     // W <- R5
20: BTFSC    R7 0     // if R7[0] == 0 then skip
21: ADDWF    R3 1     // R3 <- R3 + W
22: RRF      R3 1     // R3 <- R3 >> 1
23: RRF      R7 1     // R7 <- R7 >> 1
24: DECFSZ   R6 1     // R6 <- R6 - 1
                         and if R6 == 0 then skip
25: GOTO     LABEL1
26: MOVF     R3 0     // W <- R3
27: MOVWF    R7 1     // R7 <- W
28: MOVLW    32       // W <- 32
29: MOVWF    R6 1     // R6 <- W
30: MOVF     R5 0     // W <- R5
31: LABEL0
32: BTFSC    R7 0     // if R7[0] == 0 then skip
33: ADDWF    R2 1     // R2 <- R2 + W
34: RRF      R2 1     // R2 <- R2 >> 1
35: RRF      R7 1     // R7 <- R7 >> 1
36: DECFSZ   R6 1     // R6 <- R6 - 1
                         and if R6 == 0 then skip
37: GOTO     LABEL0
38: ADDWF    R3 1     // R3 <- R3 + W
39: MOVF     R2 0     // W <- R2
40: ADDWF    R0 1     // R0 <- R0 + W
41: MOVF     R3 0     // W <- R3
42: ADDWF    R0 1     // R0 <- R0 + W
```

図 7.13　例題 A1（$f(x) = x^4 + x^3 + x^2 + x$）における初期プログラムと進化後のプログラム（2/2）

する必要があった（初期プログラム 12 行目から 15 行目）．これに対し，進化後のプログラムは XORWF 命令によって XOR 演算を行っており，計算結果をそのまま出力値として扱うことが可能になり，偶奇判定なしに直接出力レジスタの R0 レジスタに代入することが可能になっている（進化後プログラム 9 行目と 10 行目）．これにより，初期プログラムから R6 レジスタの初期化分，および偶奇判定分の命令が削減されており，初期よりも小さなプログラムサイズが実現されている．

初期プログラム

```
 1: CLRF     R6 1     // R6 <- 0
 2: MOVF     R1 0     // W <- R1
 3: ADDWF    R6 1     // R6 <- R6+W
 4: MOVF     R2 0     // W <- R2
 5: ADDWF    R6 1     // R6 <- R6+W
 6: MOVF     R3 0     // W <- R3
 7: ADDWF    R6 1     // R6 <- R6+W
 8: MOVF     R4 0     // W <- R4
 9: ADDWF    R6 1     // R6 <- R6+W
10: MOVF     R5 0     // W <- R5
11: ADDWF    R6 1     // R6 <- R6+W
12: MOVLW    0        // W <- 0
13: BTFSC    R6 0     // if R6 & 0x0001
14: MOVLW    1        // W <- 1
15: MOVWF    R0  1    // R0 <- W
```

進化後プログラム

```
 1: XORWF    R1 0     // W <- R1 XOR W
 2: XORWF    R5 1     // R5 <- R5 XOR W
 3: MOVF     R2 0     // W <- R2
 4: XORWF    R5 1     // R5 <- R5 XOR W
 5: MOVF     R3 0     // W <- R3
 6: XORWF    R5 1     // R5 <- R5 XOR W
 7: MOVF     R4 0     // W <- R4
 8: XORWF    R5 1     // R5 <- R5 XOR W
 9: MOVF     R5 0     // W <- R5
10: IORWF    R0 1     // R0 <- R0 OR W
```

図 7.14　例題 E5（5bit-Parity）における初期プログラムと進化後のプログラム

7.4　関数同定問題

7.4.1　実験内容

本実験では，TAGP$^+$ と同期進化型 GP として個体の評価を並列に実行することで容易に並列化が可能な $(\mu+\lambda)$ 選択を用いる GP（以下，$(\mu+\lambda)$-GP），従来法である非同期 steady-state GP[17]（以下，ASSGP）を比較する．

例題としては，表 7.13 に示す 8 種類の関数を扱う．

本実験では，並列計算環境において，(**Case1**) 全個体が同一の評価速度を持つ場合，(**Case2**) 各個体の計算速度にばらつきがある場合，(**Case3**) 評価が完了しない（評価に失敗する）個体が含まれる場合，および (**Case4**)Case2 と Case3 が同時に起こる場合を想定し，ARE-GP と $(\mu+\lambda)$-GP を比較する．以下，それぞれの環境設定について説明する．

表 7.13　関数同定問題の例題（6.1 章より再掲）

R1	$f(x) = x^4 + x^3 + x^2 + x$
R2	$f(x) = x^5 - 2x^3 + x$
R3	$f(x) = x^6 - 2x^4 + x^2$
R4	$f(x, y) = x^y$
R5	$f(x) = sin(x^2) \times cos(x) - 1$
R6	$f(x) = sin(x) + sin(x + x^2)$
R7	$f(x) = ln(x + 1) + ln(x^2 + 1)$
R8	$f(x, y) = sin(x) + sin(y^2)$

Case1：全個体が同一の評価速度を持つ場合

Case1 では，全個体が単位時間あたりに同一の命令数（100 命令）を実行可能であり，かつ全個体は必ず評価が完了する．ここで，「単位時間あたりの実行可能命令数」とは 1 単位時間あたりに実行可能な命令数を表す．例えば，単位時間あたりの実行可能命令数が 100 命令であり，命令数（プログラムサイズ）が 100 命令の個体の場合には 1 単位時間ですべての命令を実行可能である．例題のデータ数が 100 個であることから 100 単位時間で 1 回の評価が完了できることを意味する．なお，本実験では 100 個体の並列評価を想定し，$(\mu + \lambda)$-GP では 1 世代進むごとにその世代における最大経過単位時間が経過，ARE-GP ではリーパー削除による評価の打ち切りも含めた並列の評価時間が経過するものとする．

Case2：各個体の計算速度にばらつきがある場合

Case1 では，全個体が一律に単位時間あたりに 100 命令を実行できるのに対し，Case2 では各個体の計算速度にばらつきを持たせる．具体的には，表 7.14 に示す割合で 100 個体中 3 個体は単位時間あたり 20 命令，3 個体は 40 命令，20 個体は 60 命令，43 個体は 80 命令，残りの 31 個体は 100 命令をそれぞれ実行できるものとする．この設定により，最も評価時間の短い個体（単位時間あたり 100 命令）に比べ，最も評価時間の長い個体（単位時間あたり 20 命令）は 5 倍の時間が必要になる．この環境は，並列計算環境において，各計算機の計算速度が異なる状況に相当する．

Case3：評価が完了しない個体が含まれる場合

Case1 では，全個体が必ず評価を完了したのに対し，Case3 ではある一定の割合 P_f の個体が評価が完了しない．本実験では，$P_f = 0.05$ に設定し，全評価の 5% が完了しない状況を模擬する．具体的には，進化の過程において常にランダムな 5% の個体のプログラ

表 7.14　Case2 における単位時間あたりの実行可能命令のばらつき

単位時間あたりの実行可能命令数	個体数
20	3
40	3
60	20
80	43
100	31

ムカウンタが増加しないようにし，プログラムの実行が完了しないようにする．同期進化である $(\mu + \lambda)$-GP では全個体の評価値を算出する必要があるため，評価の待機時間に上限を設け，評価を打ち切る必要がある．本実験では，評価時間が設定した待機単位時間（詳細は 7.4.3 章）を超過した場合は，その個体の評価値を $-\infty$ に設定する．この環境は，通信エラーや計算機の故障によって計算結果が得られない状況や解自体の性質（例えば，無限ループ）によって計算が終了しない状況に相当する．

Case4：Case2 と Case3 が同時に起こる場合

Case4 では，Case2 と Case3 が同時に起こる場合を想定する．具体的には，単位時間あたりの実行可能命令数は図 7.14 によって決定し，P_f は 0.05 に設定する．

7.4.2　評価基準と設定

適合度関数は式 (7.7) に示す平均二乗誤差を用いる．

$$f_{reg} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y_i} - y_i^*)^2 \tag{7.7}$$

ここで，$\hat{y_i}$ は i 番目の入力に対するプログラムの出力結果，y_i^* は i 番目の入力に対する目標値，n はデータ数（$= 100$）を表す．

本実験では，並列計算環境において各個体の評価時間が異なる環境を模擬した実験を行い，同一評価回数における比較と，同一経過時間における比較の 2 種類の尺度で TAGP$^+$ と同期 GP の $(\mu + \lambda)$-GP，従来の非同期 GP の ASSGP を比較する．評価回数は 1.0×10^6 回，経過時間は 1.0×10^7 単位時間を上限とする．実験は各ケース 30 試行ずつ行い，集団内の最小適合度の平均をもとに評価する．

パラメータ設定は TAGP$^+$，$(\mu + \lambda)$-GP，ASSGP のすべての手法で共通に表 7.15 に示す値を使用する．TAGP$^+$ において，適合度スケーリングと P_{down}^{α} の調整に用いるパラメータ N_f と N_{update} はともに 100 に設定する．

表 7.15　実験パラメータ

最大命令数	256 命令	突然変異率	0.1
集団サイズ	100	命令挿入率	0.1
交叉率	0.7	命令削除率	0.1
交叉法	二点交叉		

7.4.3　結果

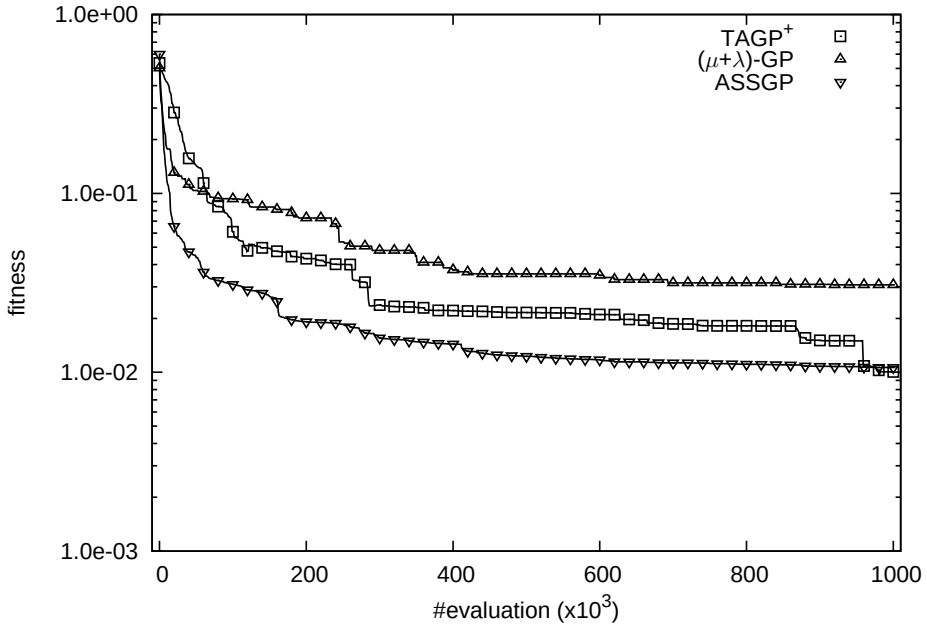
Case1：全個体が同一の評価速度を持つ場合

図 7.15 に Case1 において同一の評価回数における TAGP$^+$ と $(\mu+\lambda)$-GP，ASSGP の平均適合度の変化を示す．図 7.15 において，横軸は評価回数，縦軸は 30 試行の平均適合度を示す．四角プロットは TAGP$^+$ の結果，三角プロットは $(\mu+\lambda)$-GP の結果，逆三角プロットは ASSGP の結果を表す．図 7.15 の結果から，TAGP$^+$ は同一の評価回数では $(\mu+\lambda)$-GP と比較すると同等以上の性能を示しているが，ASSGP と比較すると R1 と R8 を除く全ての例題で性能が劣ることがわかる．

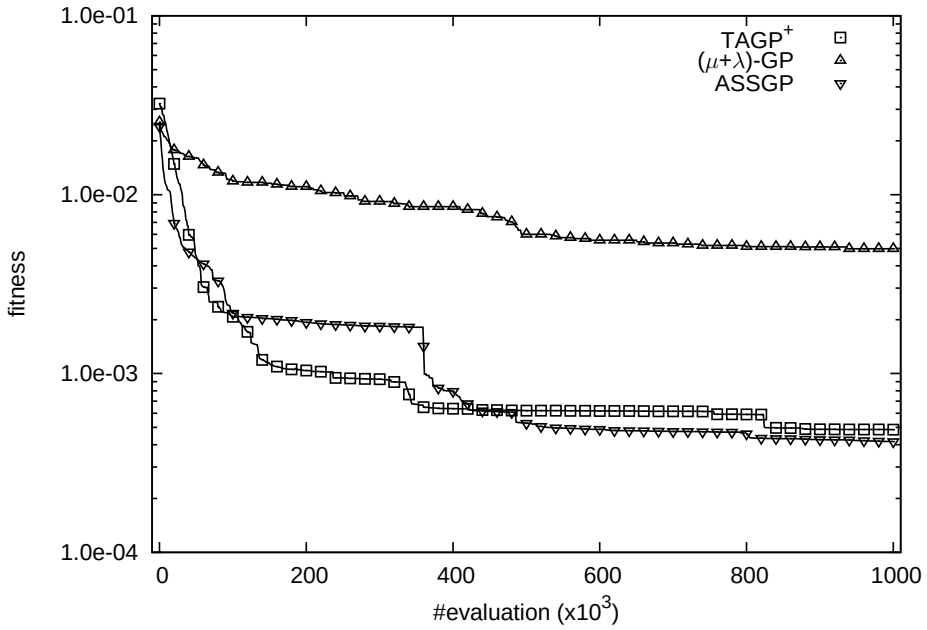
これに対し，同一の経過単位時間における TAGP$^+$ と $(\mu+\lambda)$-GP，ASSGP の平均適合度の変化を図 7.16 に示す．図 7.16 において横軸は経過単位時間，縦軸は 30 試行の平均適合度を示し，四角プロットが TAGP$^+$ の結果，三角プロットが $(\mu+\lambda)$-GP の結果，逆三角プロットは ASSGP の結果を表す．

結果から，同一の経過時間で TAGP$^+$ と同期 EA の $(\mu+\lambda)$-GP を比較した場合，同一の評価回数で比較した場合に比べて差が顕著になっていることがわかる．これは，$(\mu+\lambda)$-GP が評価時間の長い個体を待機してからでないと子個体を生成できないのに対し，TAGP$^+$ は他の個体の評価を待機する必要なく進化を継続可能であるため，単位時間あたりに効率的な進化が可能なためである．また，ASSGP と比較すると，いずれの例題においても収束速度，最終世代での平均的適合度ともに優れていることがわかる．これは，ASSGP が非同期な進化の中で評価値の高くない個体がトーナメント選択によって親個体として選択される可能性があるのに対し，TAGP$^+$ が一定以上の評価値を持つ個体のみが親個体として選択することによって評価時間が短く評価値の低い個体が親個体として選択されるのを防いでいるためである．

以上の結果から，TAGP$^+$ は同一の評価回数では従来の非同期 EA である ASSGP に比べ性能が劣るのの，同一の経過時間では $(\mu+\lambda)$-GP，ASSGP を上回る解探索性能を有しているといえる．

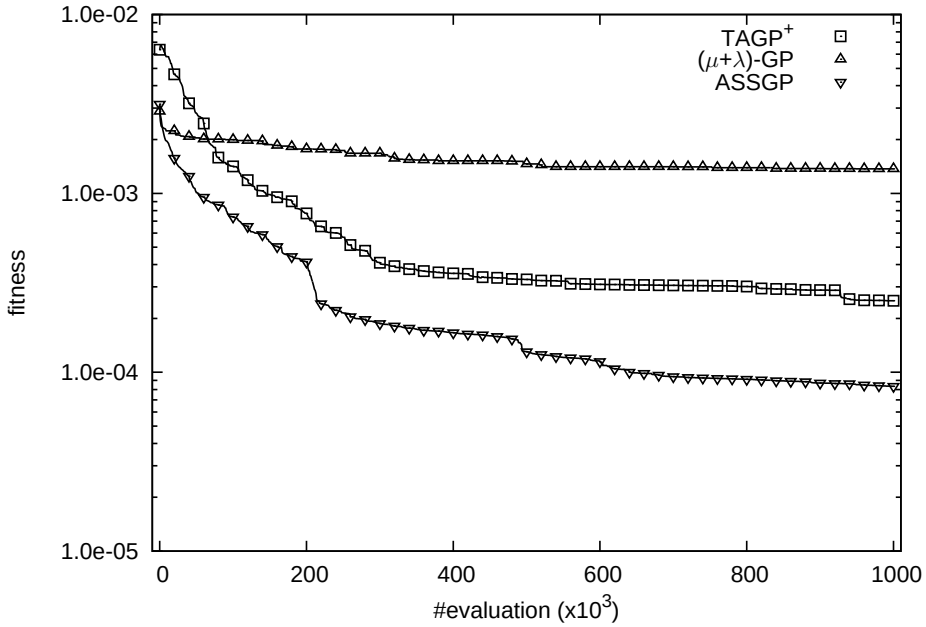


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

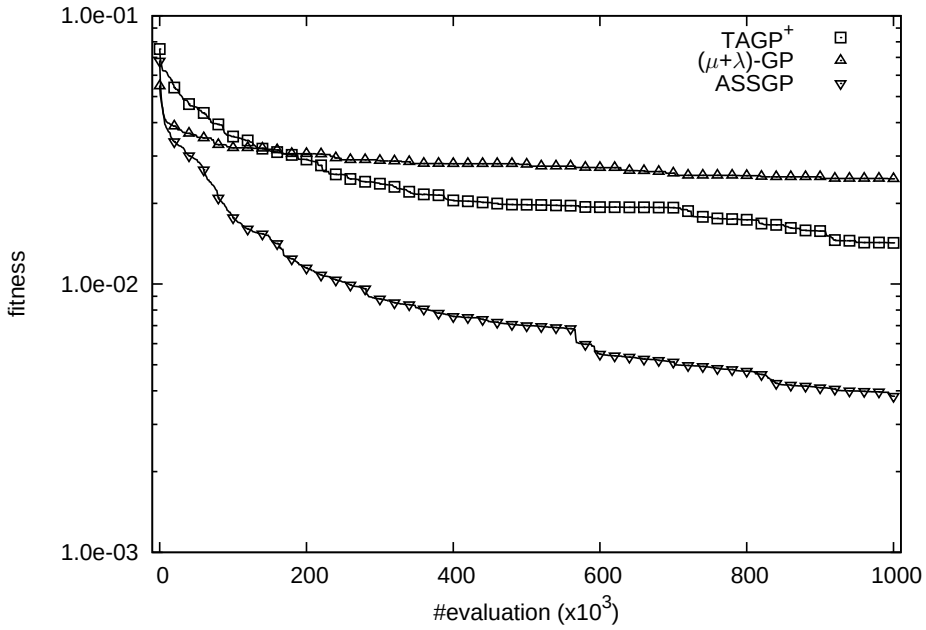


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 7.15　Case1：同一の評価回数における平均適合度の変化（1/4）

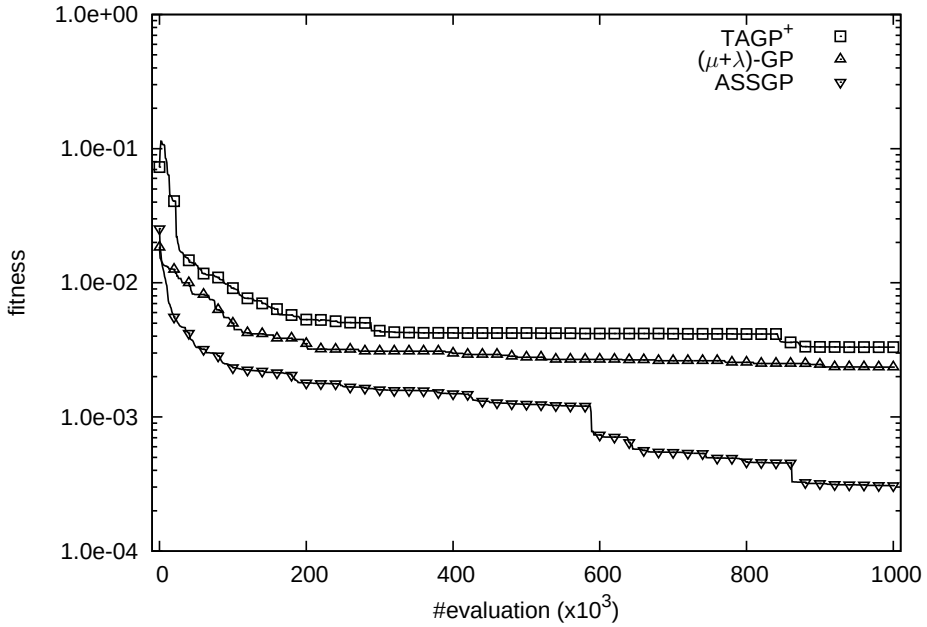


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

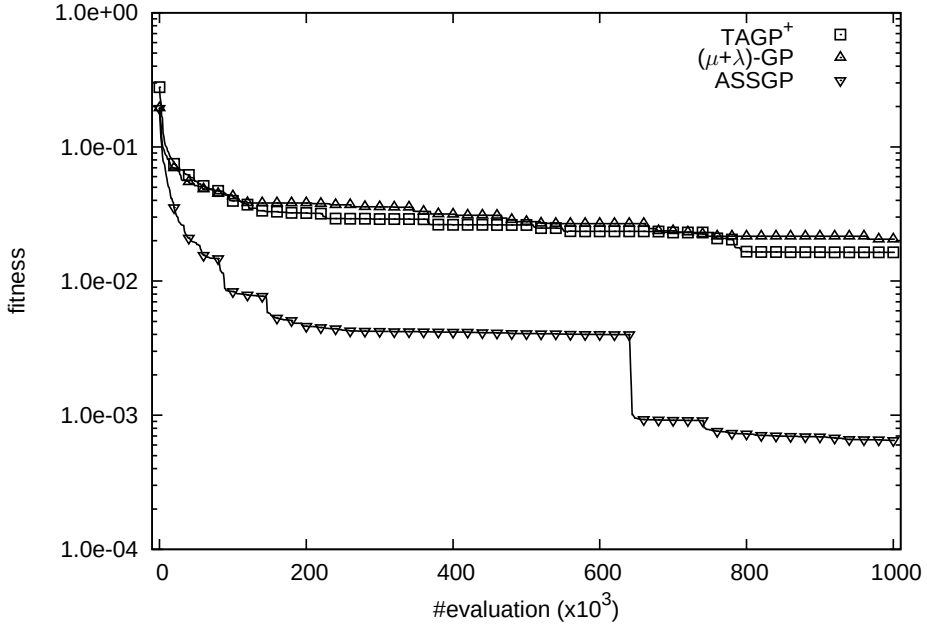


(d) R4（$f(x, y) = x^y$）

図 7.15　Case1：同一の評価回数における平均適合度の変化（2/4）

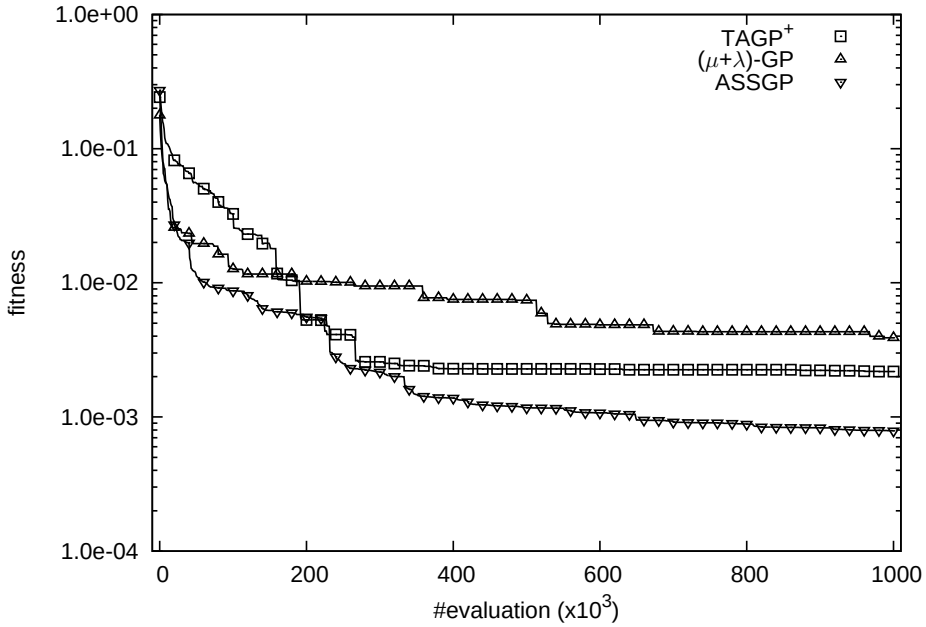


(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

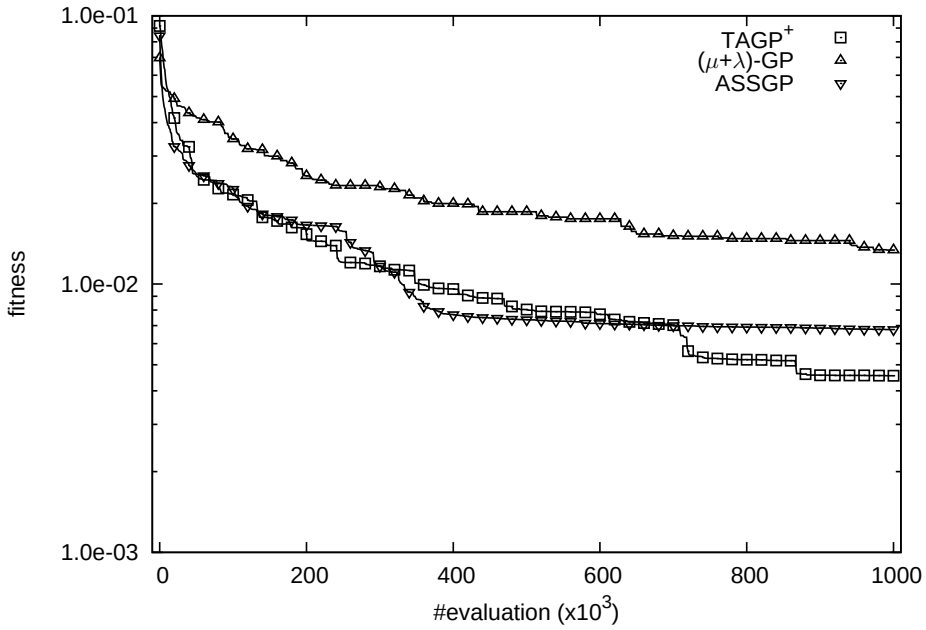


(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 7.15　Case1：同一の評価回数における平均適合度の変化（3/4）

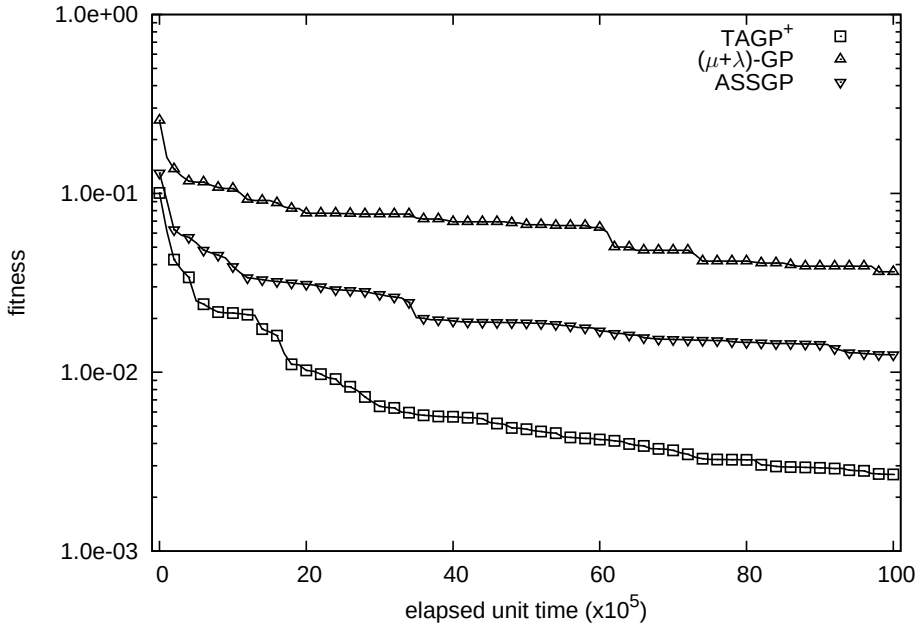


(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）

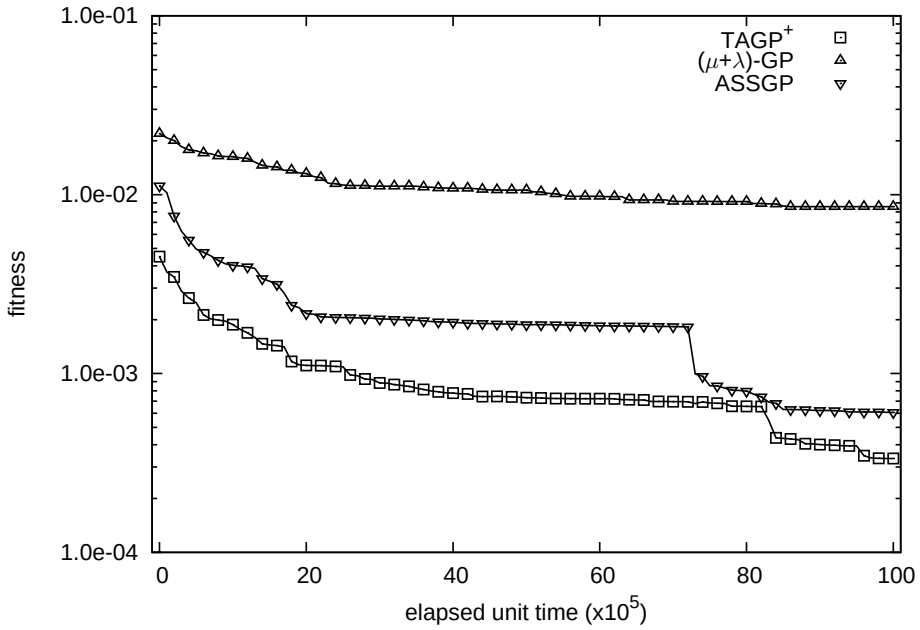


(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 7.15　Case1：同一の評価回数における平均適合度の変化（4/4）

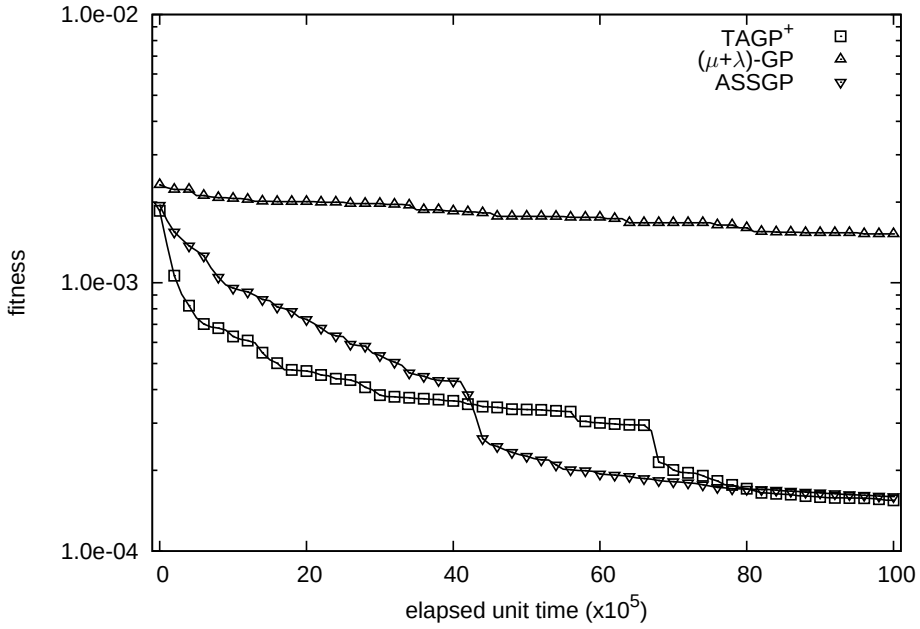


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

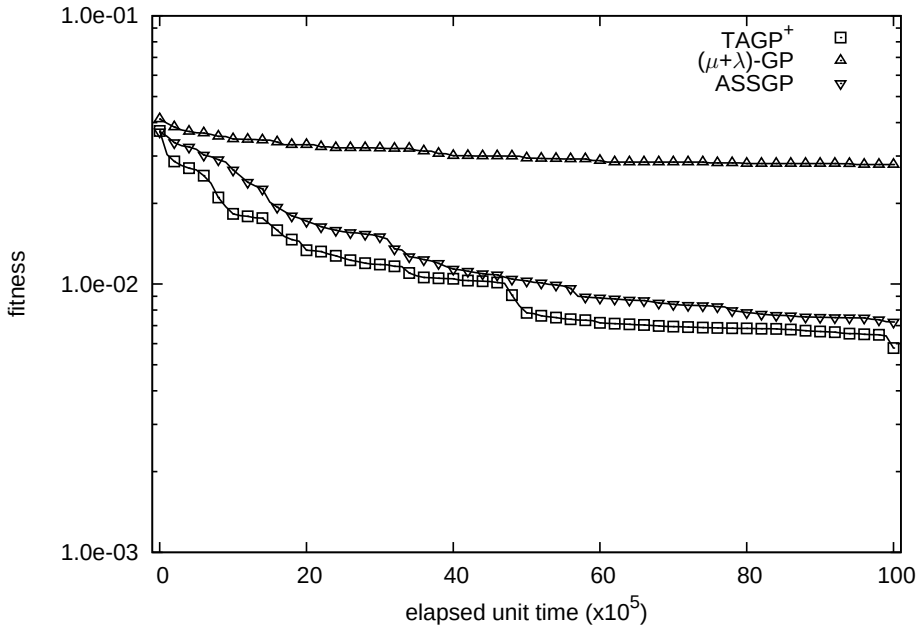


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 7.16　Case1：同一の経過単位時間における平均適合度の変化（1/4）

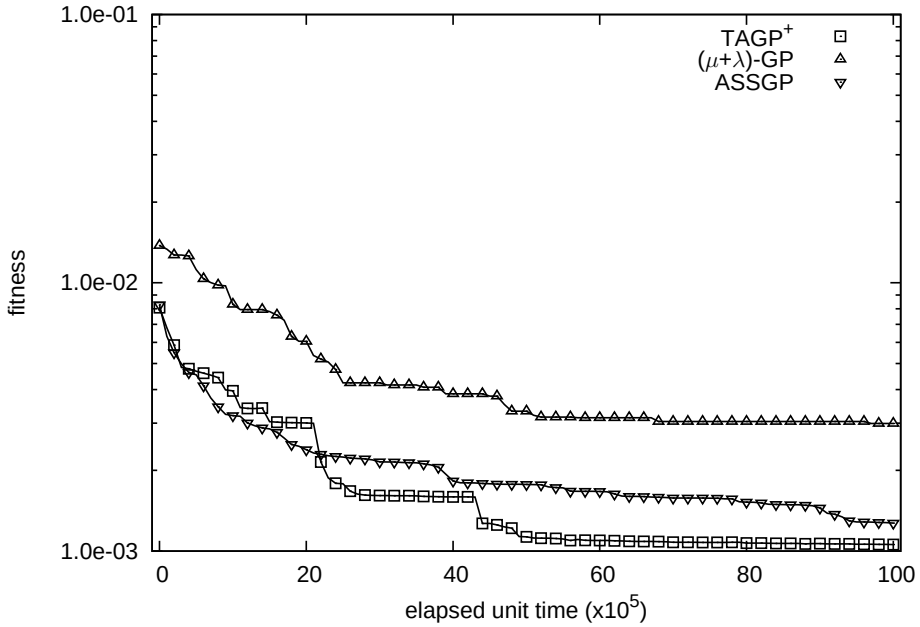


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

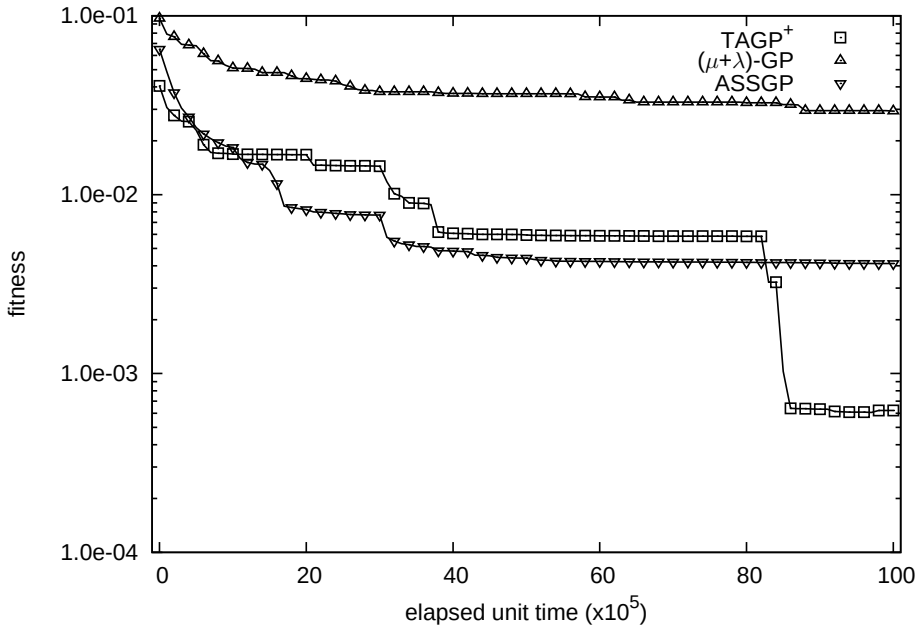


(d) R4（$f(x, y) = x^y$）

図 7.16　Case1：同一の経過単位時間における平均適合度の変化（2/4）

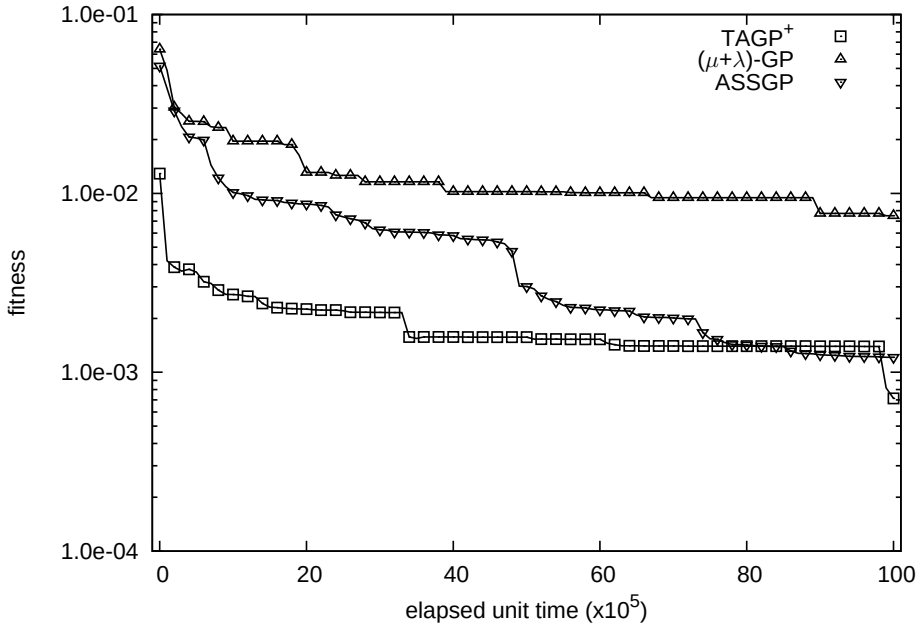


(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

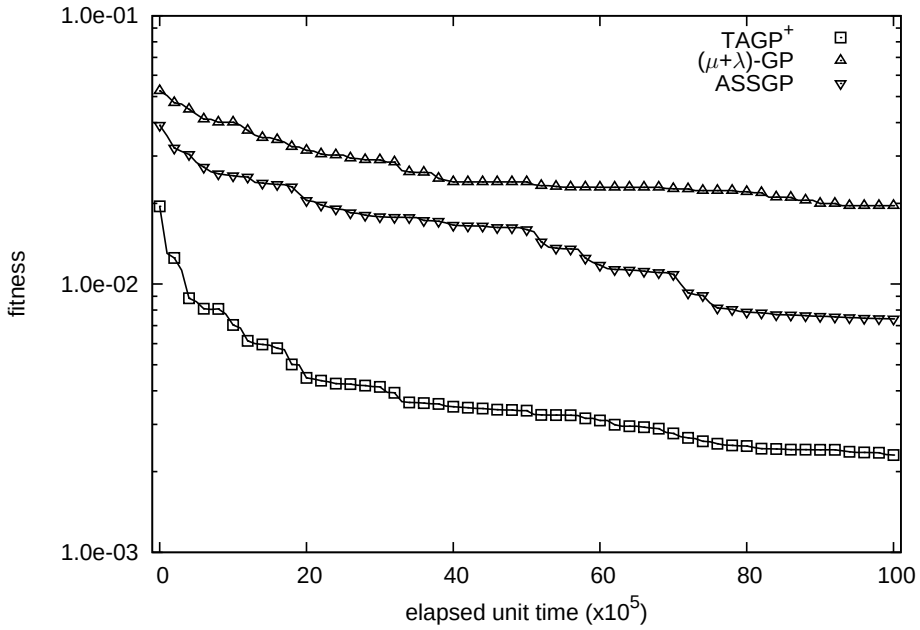


(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 7.16　Case1：同一の経過単位時間における平均適合度の変化（3/4）

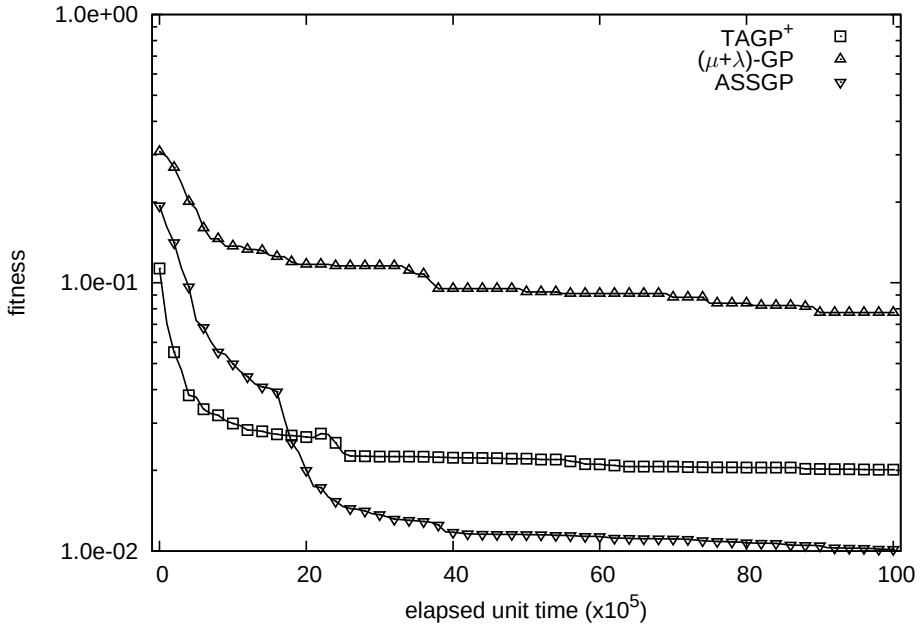


(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）

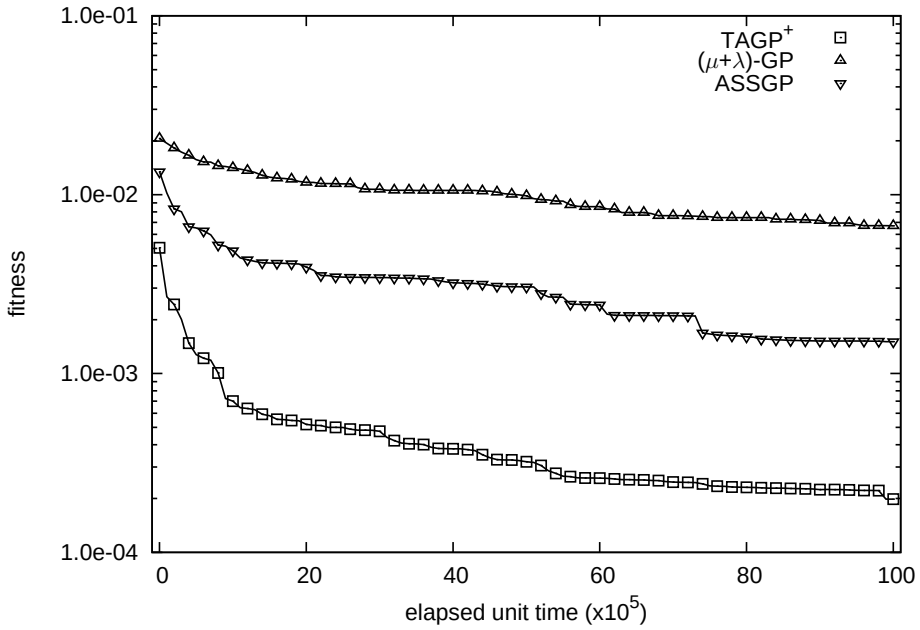


(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 7.16　Case1：同一の経過単位時間における平均適合度の変化（4/4）

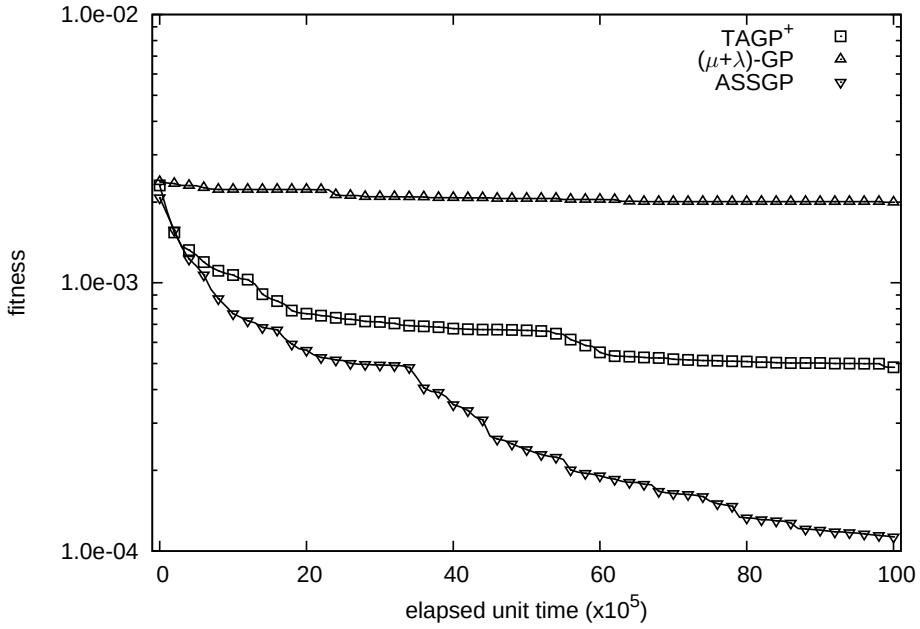


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

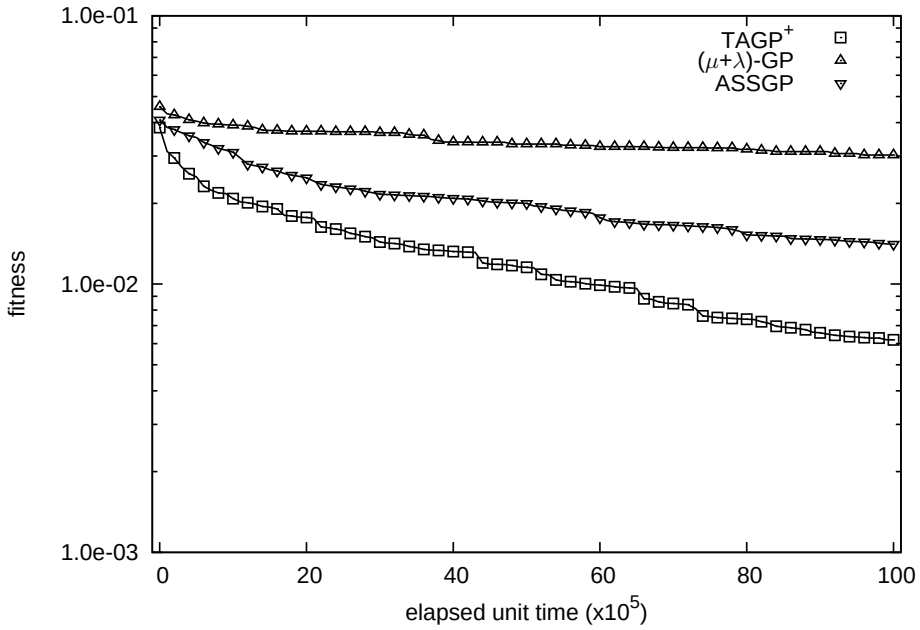


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 7.17　Case2：同一の経過単位時間における平均適合度の変化（1/4）

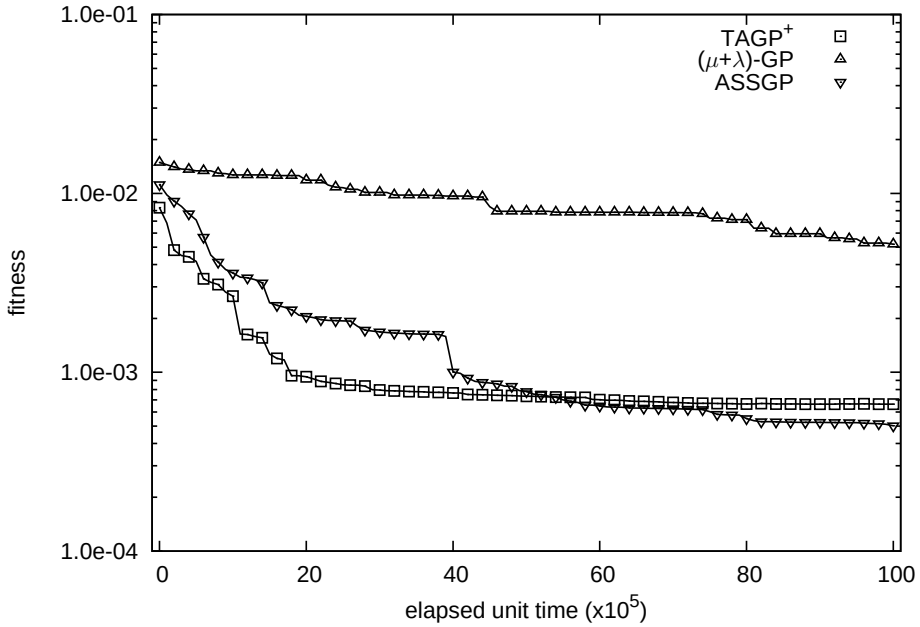


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

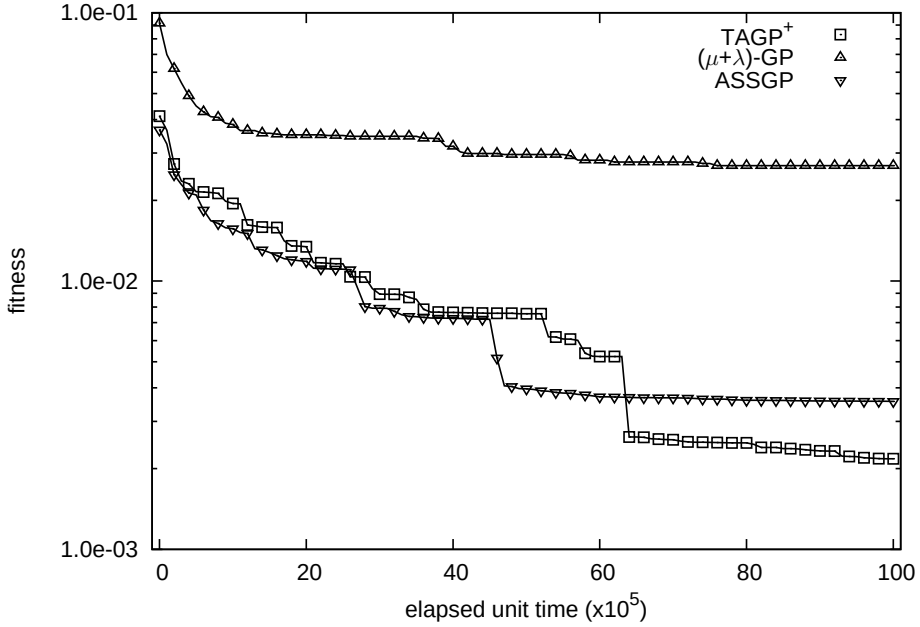


(d) R4（$f(x, y) = x^y$）

図 7.17　Case2：同一の経過単位時間における平均適合度の変化（2/4）

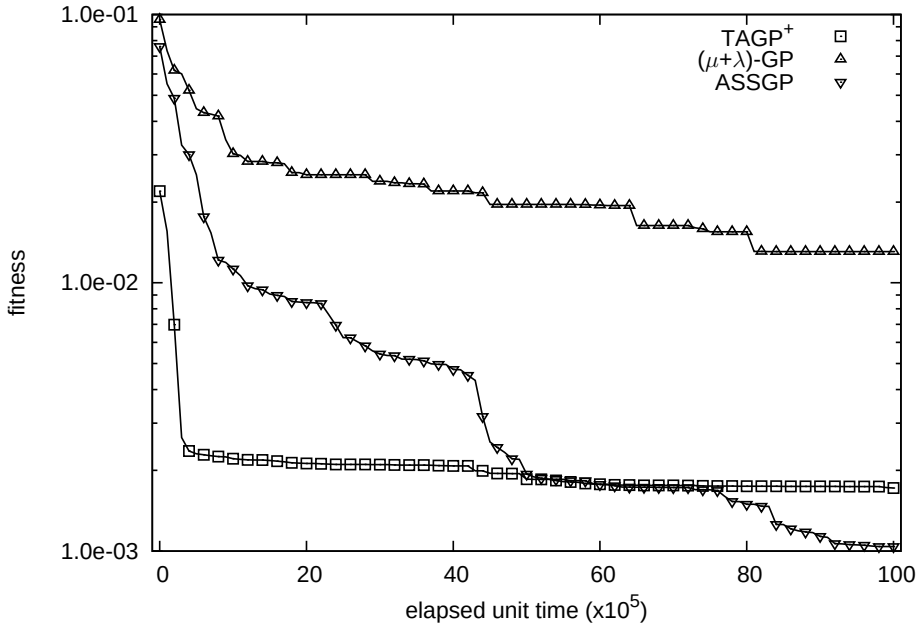


(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

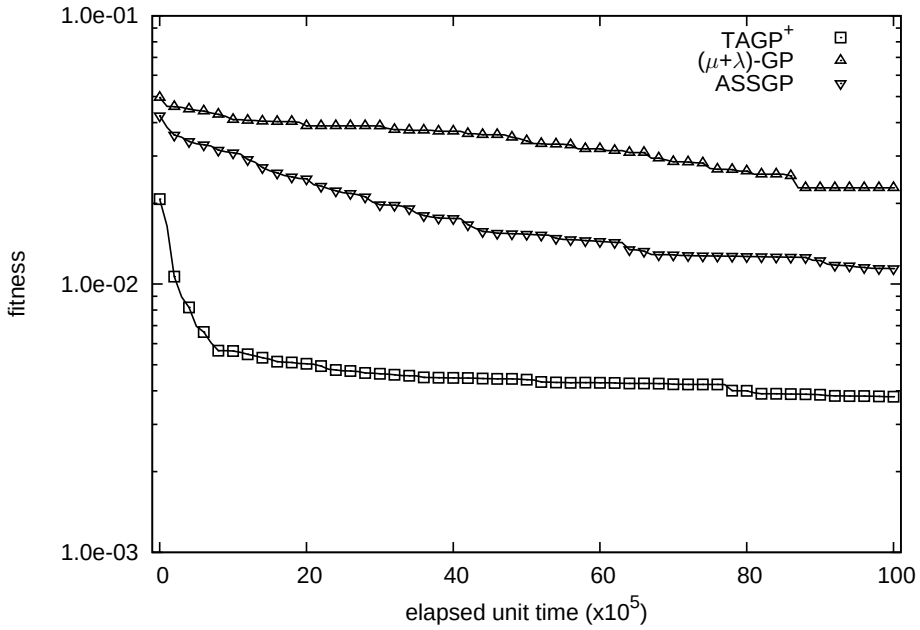


(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 7.17　Case2：同一の経過単位時間における平均適合度の変化（3/4）



(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）



(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 7.17　Case2：同一の経過単位時間における平均適合度の変化（4/4）

Case2：各個体の計算速度にばらつきがある場合

各個体の計算速度にばらつきのある Case2 の同一経過単位時間における平均適合度の変化を図 7.17 に示す．図 7.17 において，各軸，各プロットは図 7.16 と同一である．

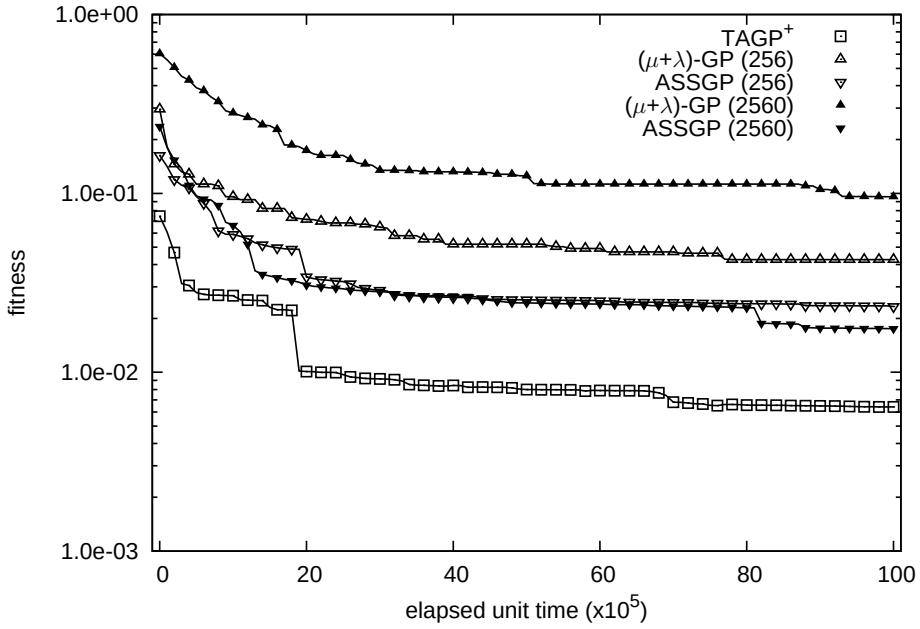
まず，$(\mu+\lambda)$-GP の結果に着目すると，いずれの例題においても他の手法に比べて性能が劣ることが確認できる．これは，最も単位時間あたりの実行可能命令数が少ない 20 命令/単位時間の個体の評価を待ってから次世代の子個体を生成するため，1 世代に最悪で 5 倍の評価時間が必要となるため，同期進化によって評価時間の無駄が増加するためである，これに対し，TAGP$^+$ と ASSGP を比較すると，Case1 ではすべての例題で TAGP$^+$ が ASSGP を上回る性能を示していたのに対し，Case2 では R1，R3，R5，R7 で ASSGP の性能を下回っていることがわかる．特にこれらの例題では，TAGP$^+$ は進化の初期段階では ASSGP を上回る性能を示しているものの，その後進化が停滞していることが確認できる．これは，進化の初期段階で局所解に陥り，そこを脱することができないためである．このことから，TAGP$^+$ は計算時間にばらつきがある場合に性能が低下する可能性があることが明らかになった．

Case3：評価が完了しない個体が含まれる場合

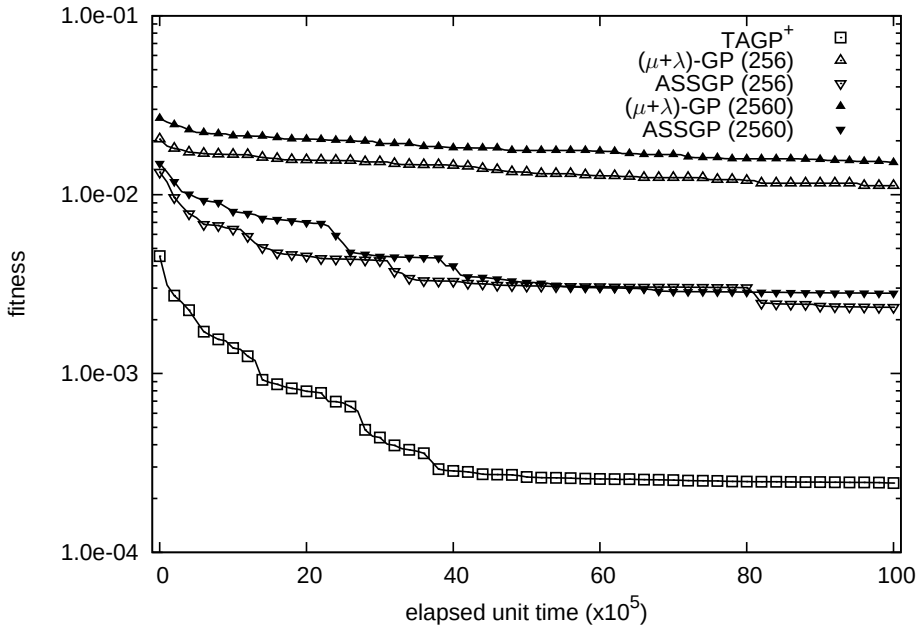
評価が完了しない個体が含まれる Case3 の同一の経過単位時間における平均適合度の変化を図 7.18 に示す．図 7.18 において，各軸，各プロットは図 7.16 と同一である．本実験では，個体（プログラム）中の最大命令数を 256 命令，単位時間あたりの実行可能命令数を 100 命令，データ数を 100 個としているため，最適な待機時間は 256 単位時間となる．各例題において，最大待機時間は 256 単位時間に設定しており，その時間を超えた個体の評価値は $-\infty$ とする．なお比較のために，最適値の 10 倍に当たる 2560 単位時間の結果も示している（図 7.18 中の塗りつぶしのプロット）．

まず，2 種類の上限値を設定した結果から，$(\mu+\lambda)$-GP は上限値の設定によって解探索性能が著しく低下しており，ASSGP も $(\mu+\lambda)$-GP と比較すると影響は小さいものの特に探索の初期において性能が低下していることがわかる．このことから，$(\mu+\lambda)$-GP，ASSGP では，上限値の設定が性能に影響を与えることが確認できる．これに対し，TAGP$^+$ はすべての例題において待機時間の上限を設定せずに進化を継続できており，さらに R6 以外のすべての例題で上限値を最適に設定した $(\mu+\lambda)$-GP，ASSGP を上回る性能を有していることがわかる．

これらの結果から，TAGP$^+$ は評価が完了しない個体が含まれる場合でも従来のように待機時間の上限を必要とせずに従来を上回る性能を実現できることが明らかになった．

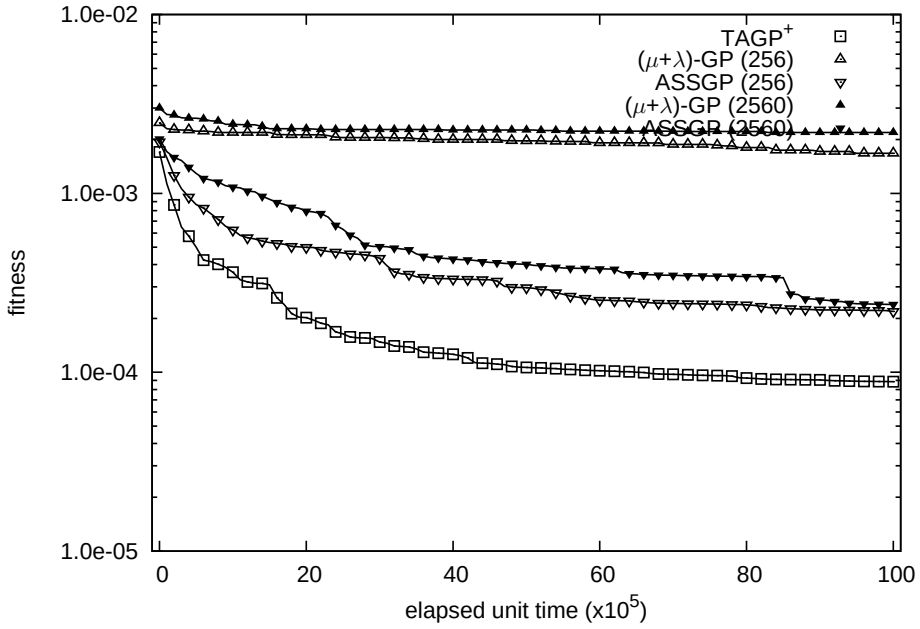


(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）

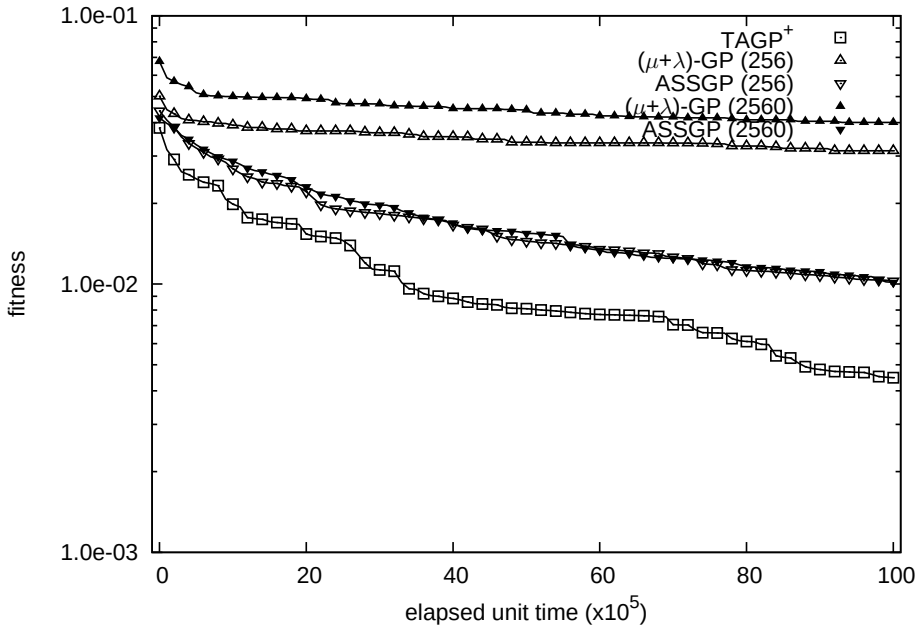


(b) R2（$f(x) = x^5 - 2x^3 + x$）

図 7.18　Case3：同一の経過単位時間における平均適合度の変化（1/4）

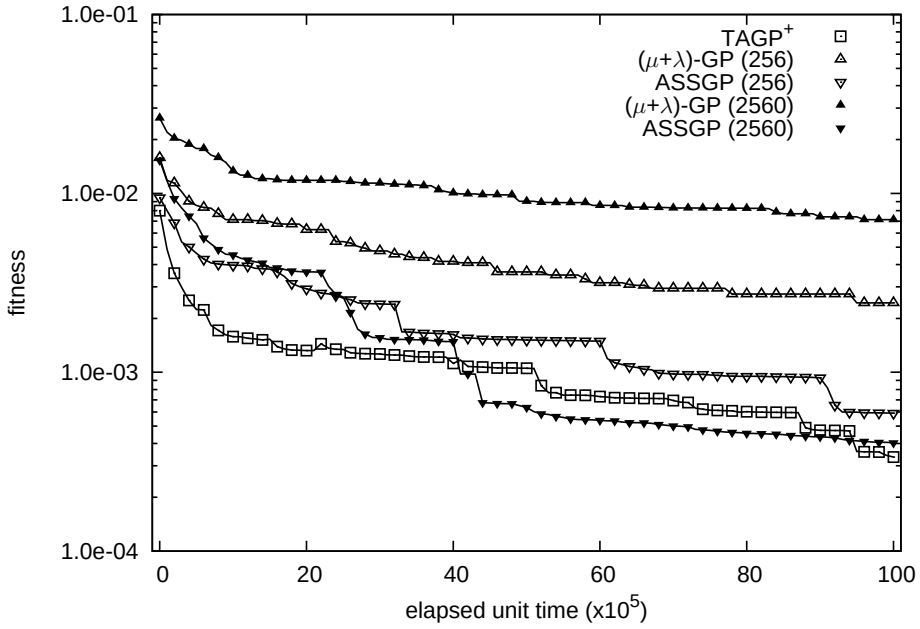


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

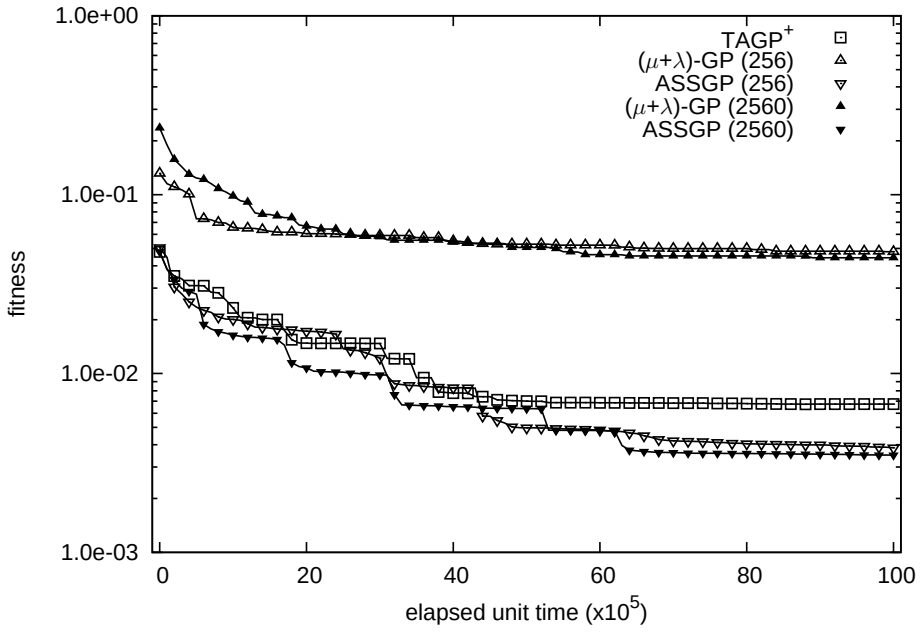


(d) R4（$f(x, y) = x^y$）

図 7.18　Case3：同一の経過単位時間における平均適合度の変化（2/4）

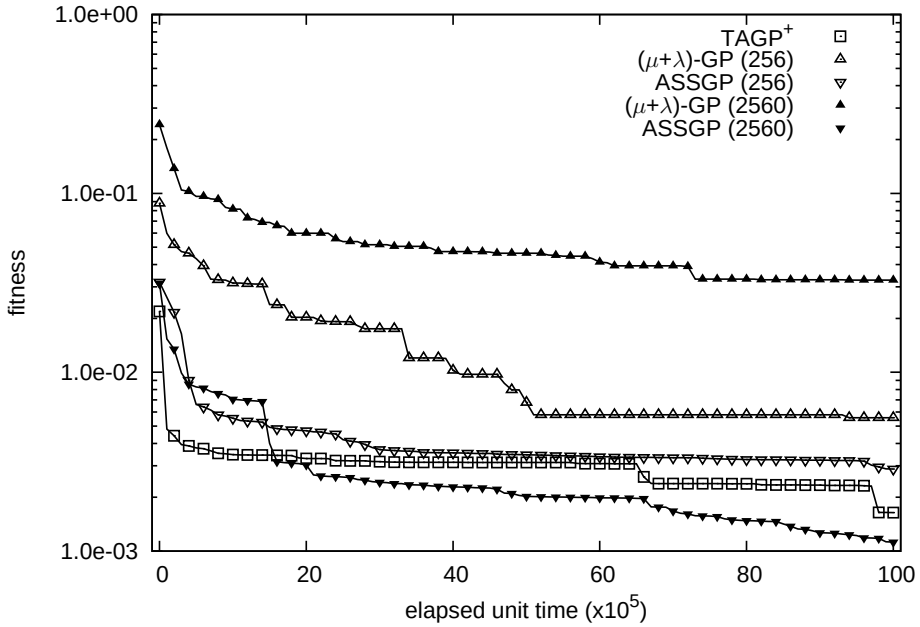


(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）

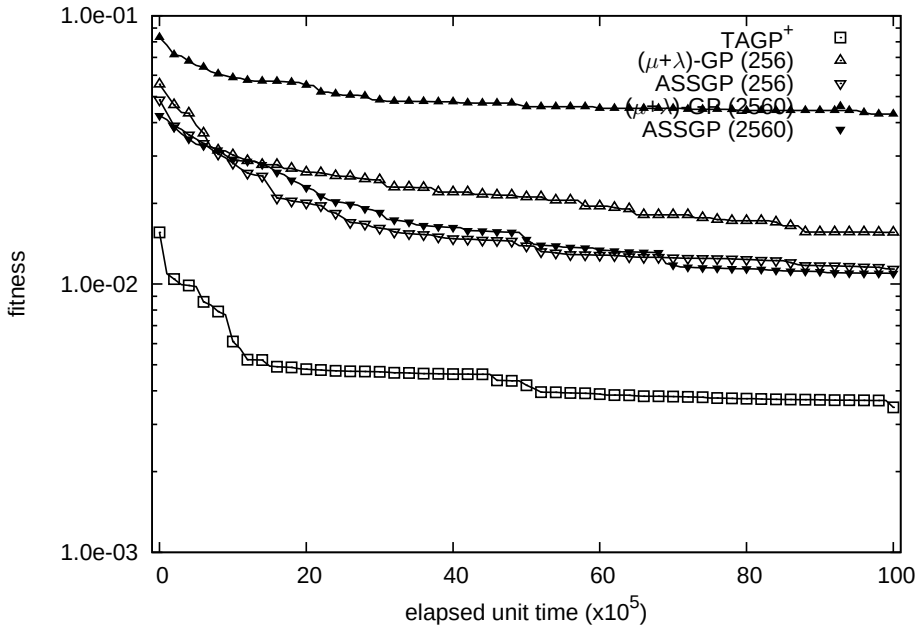


(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

図 7.18　Case3：同一の経過単位時間における平均適合度の変化（3/4）

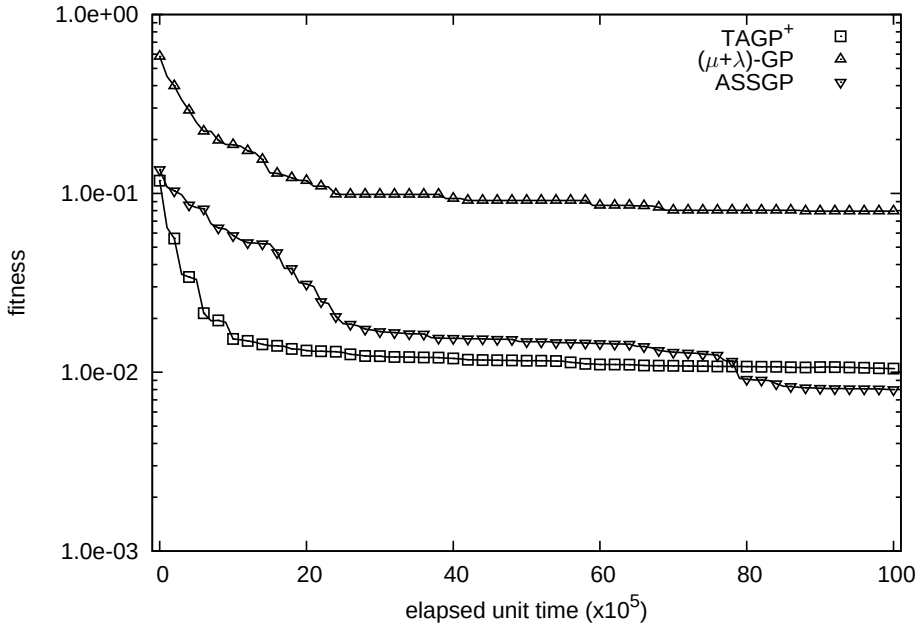


(g) R7 ($f(x) = ln(x+1) + ln(x^2+1)$)

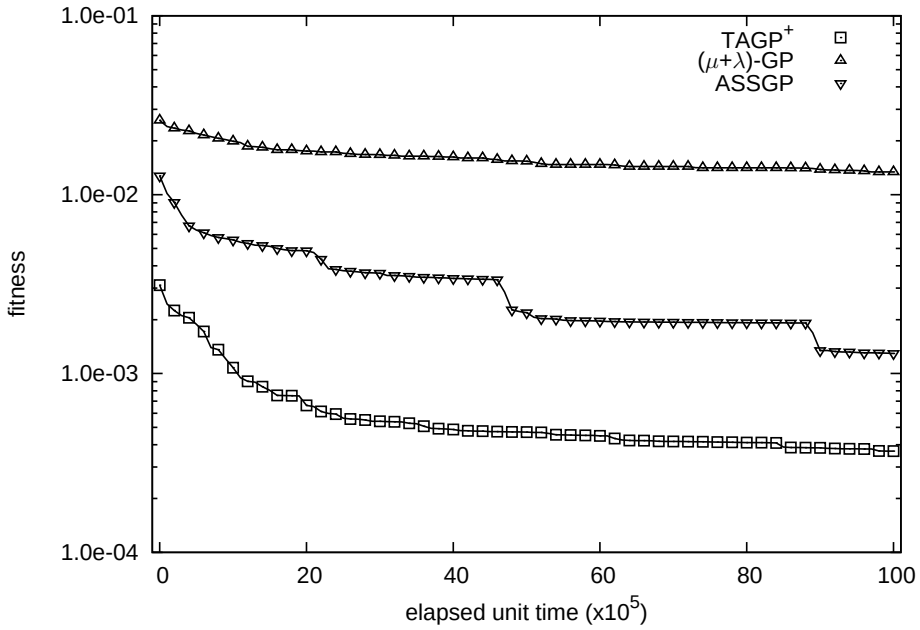


(h) R8 ($f(x,y) = sin(x) + sin(y^2)$)

図 7.18　Case3：同一の経過単位時間における平均適合度の変化（4/4）

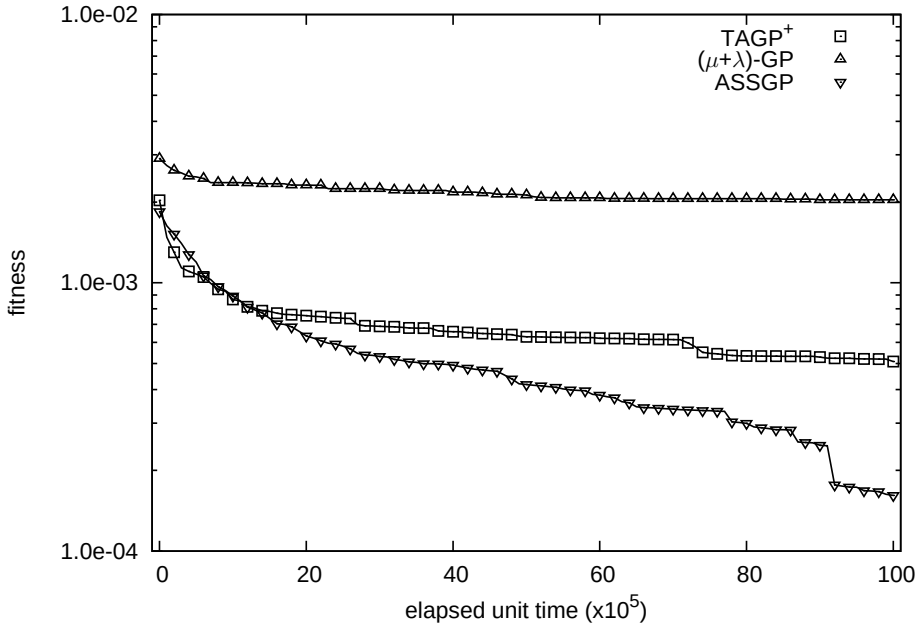


(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）

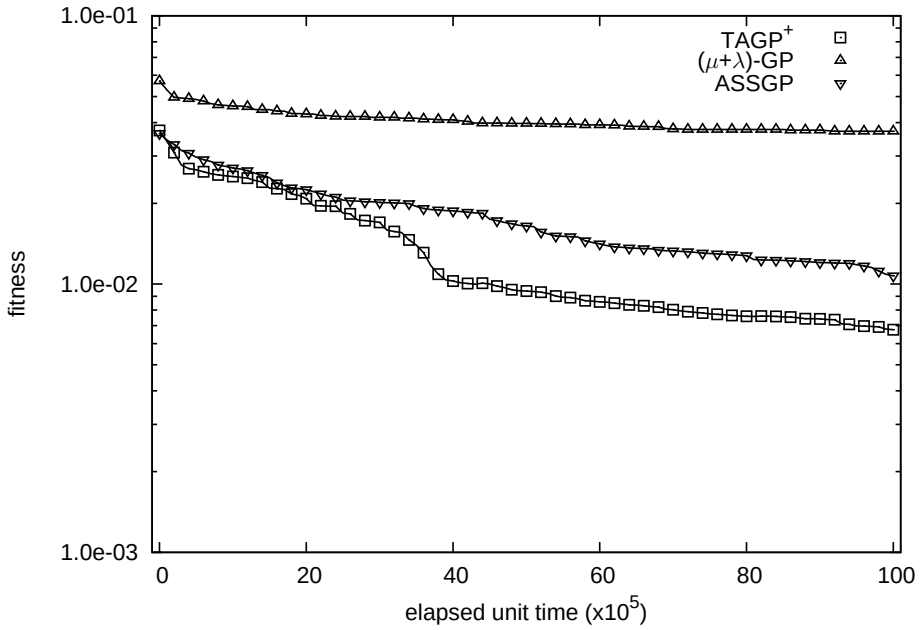


(b) R2（$f(x) = x^5 - 2x^3 + x$）

図 7.19　Case4：同一の経過単位時間における平均適合度の変化（1/4）

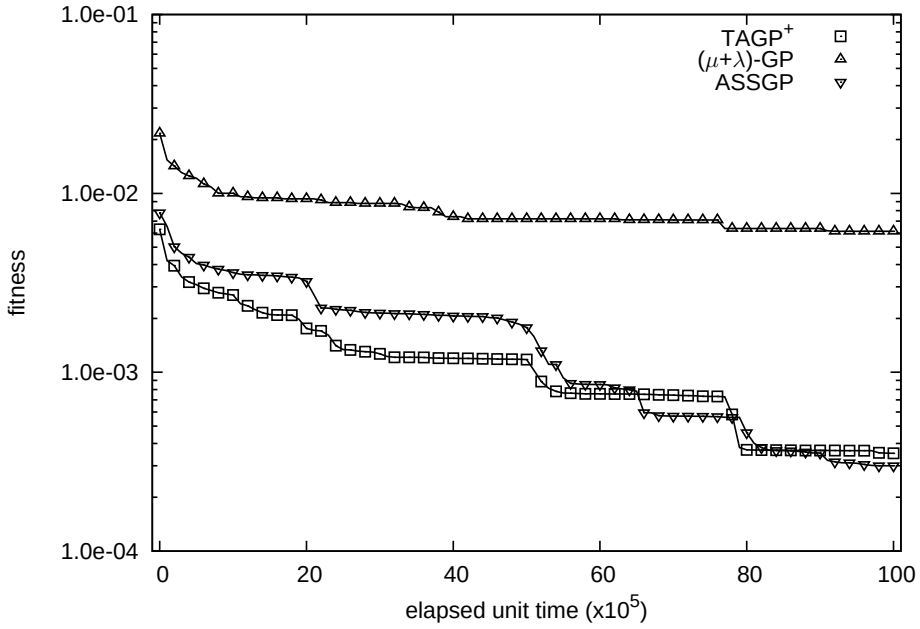


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

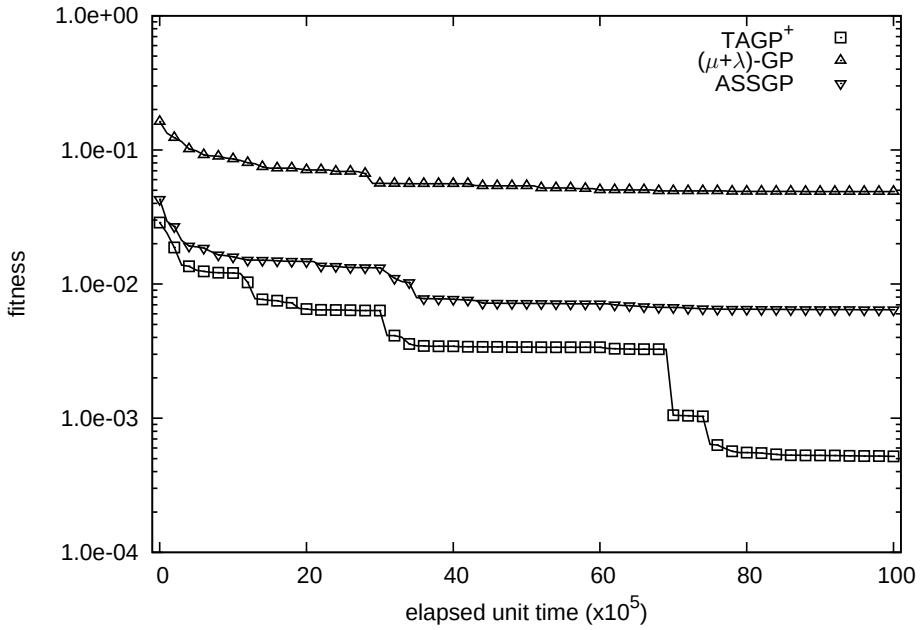


(d) R4（$f(x, y) = x^y$）

図 7.19　Case4：同一の経過単位時間における平均適合度の変化（2/4）

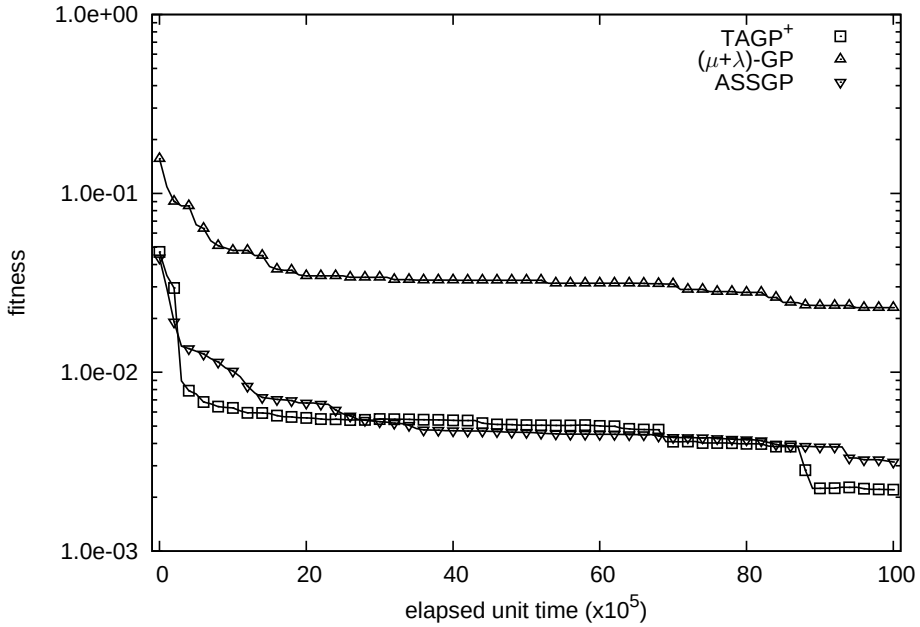


(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）

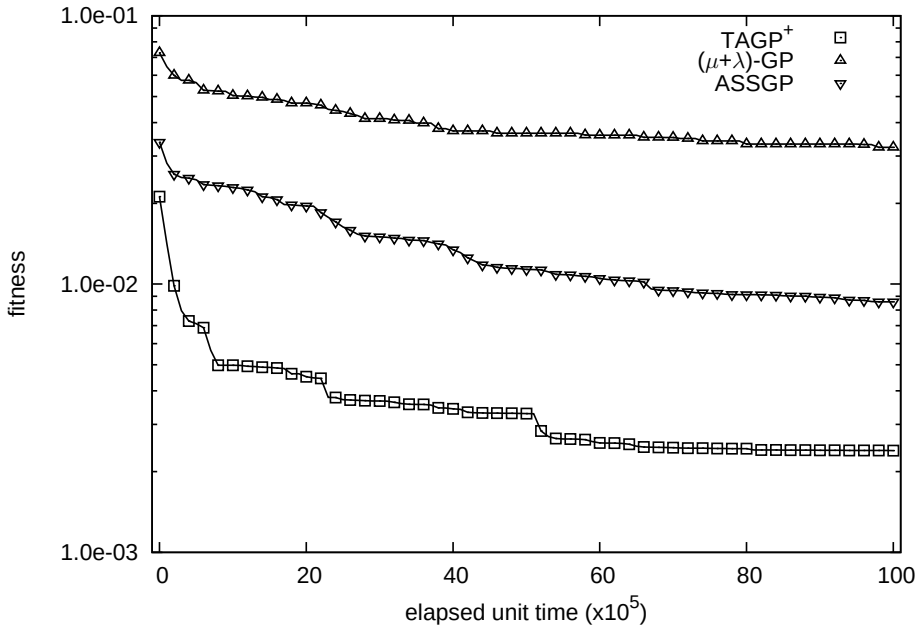


(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

図 7.19　Case4：同一の経過単位時間における平均適合度の変化（3/4）



(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）



(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 7.19　Case4：同一の経過単位時間における平均適合度の変化（4/4）

Case4：Case2 と Case3 が同時に起こる場合

Case2 と Case3 が同時に起こる Case4 の同一の経過単位時間における平均適合度の変化を図 7.19 に示す．図 7.19 において，各軸，各プロットは図 7.16 と同一である．本実験では各個体の計算速度にばらつきがあるため，$(\mu + \lambda)$-GP，ASSGP における待機時間は単位時間あたりの実行可能命令数が最も少ない 20 命令/単位時間に上限値を合わせた 1280 単位時間の設定を用いる．また，上限値を超えた個体の評価値は Case3 と同様に $-\infty$ とする．

まず，$(\mu + \lambda)$-GP はいずれの例題においても，他の手法に比べて性能が劣ることが確認できる．これは，Case2 と同様に最も評価時間の長い個体を待つ必要がある上に全体の 5% の個体の評価が完了しないため，毎世代必ず待機時間の上限（1280 単位時間）が必要となるためである．これに対し，TAGP$^+$ と ASSGP を比較すると，R2，R4，R6，R7，R8 では TAGP$^+$ の性能が ASSGP を上回っているものの，Case2 で TAGP$^+$ が ASSGP よりも劣る結果となった R1，R3，R5 において同様に TAGP$^+$ が劣る結果となっていることがわかる．しかし，R1 と R5 については最終世代の適合度に大きな差が見られないことから，これらの例題では TAGP$^+$ と ASSGP は同等の性能を有しているといえる．

以上の結果から，TAGP$^+$ は各個体の計算速度にばらつきがあり，かつ評価が完了しない個体が含まれるような同期進化では並列計算の効率が著しく低下してしまう環境においても性能を低下させることなく進化が可能であり，かつ従来の非同期 GP である ASSGP と同等以上の性能を示すことが明らかになった．

7.4.4 考察

TAGP$^+$ と $(\mu + \lambda)$-GP，ASSGP の結果の有意差を統計的に示すために，10^7 単位時間経過後の平均適合度についてノンパラメトリック検定の一つであるマン–ホイットニーの U 検定（Mann–Whitney U test） [44] を用いて検定を行った結果の P 値を表 7.16 に示す．検定の結果，$(\mu + \lambda)$-GP との比較ではすべての例題，ケースにおいて有意水準 $\alpha = 0.05$ で有意差があることが確認された．また，ASSGP との比較では，Case1 と Case3 においては多くの例題とケースで有意水準 $\alpha = 0.05$ で有意差があることが確認され，TAGP$^+$ が平均適合度で劣る Case3 の R6 では有意差が見られないことから，Case1 と Case3 では ASSGP と比較して同等以上の性能を実現できていることが確認された．一方，計算速度に差がある Case2 と Case4 においては，Case2 の R1，および Case4 の R1 と R5 を除いては TAGP$^+$ が ASSGP と比較して同等以上の性能を示すことが確認されたが，これらのケースでは ASSGP の平均適合度が有意水準 $\alpha = 0.05$ で有意に優れていることが示された．

このことから，特に計算速度に差がある場合に TAGP$^+$ の性能が低下する可能性が示唆

表 7.16　各実験ケースにおける 10^7 単位時間経過後の平均適合度とマン–ホイットニーの U 検定によって得られた P 値（1/2）

		R1	R2	R3	R4
Case1	TAGP$^+$	$\mathbf{2.7 \times 10^{-3}}$	$\mathbf{3.3 \times 10^{-4}}$	$\mathbf{1.5 \times 10^{-4}}$	$\mathbf{5.8 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	3.6×10^{-2}	8.6×10^{-3}	1.5×10^{-3}	2.8×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	1.3×10^{-2}	6.1×10^{-4}	1.6×10^{-4}	7.2×10^{-3}
	（P 値）	**(< 0.01)**	(0.69)	(0.50)	**(0.0496)**
Case2	TAGP$^+$	2.0×10^{-2}	$\mathbf{2.0 \times 10^{-4}}$	4.8×10^{-4}	$\mathbf{6.2 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	7.7×10^{-2}	6.7×10^{-3}	2.0×10^{-3}	3.0×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	$\mathbf{1.0 \times 10^{-2}}$	1.5×10^{-3}	$\mathbf{1.1 \times 10^{-4}}$	1.4×10^{-2}
	（P 値）	**(< 0.01)**	**(0.0496)**	(0.096)	**(< 0.01)**
Case3	TAGP$^+$	$\mathbf{6.4 \times 10^{-3}}$	$\mathbf{2.4 \times 10^{-4}}$	$\mathbf{8.9 \times 10^{-5}}$	$\mathbf{4.5 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	4.3×10^{-2}	1.1×10^{-2}	1.7×10^{-3}	3.1×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	2.3×10^{-2}	2.3×10^{-3}	2.2×10^{-4}	1.0×10^{-2}
	（P 値）	(0.10)	(0.079)	**(0.045)**	**(< 0.01)**
Case4	TAGP$^+$	1.0×10^{-2}	$\mathbf{3.7 \times 10^{-4}}$	5.1×10^{-4}	$\mathbf{6.8 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	8.0×10^{-2}	1.3×10^{-2}	2.0×10^{-3}	3.7×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	$\mathbf{8.0 \times 10^{-3}}$	1.3×10^{-3}	$\mathbf{1.6 \times 10^{-4}}$	1.1×10^{-2}
	（P 値）	**(0.033)**	(0.80)	(0.46)	**(0.041)**

された．この原因を分析するために，Case2 の R1 において 30 試行の中で最も適合度の値が悪い（高い）1 試行について分析する．図 7.20 に対象となる試行の四分位数を適合度スケーリングした値の推移，図 7.21 にその時のリーパー制御パラメータ P^{α}_{down} の推移を示す．図 7.20 において横軸は経過単位時間，縦軸は適合度を表し，丸プロットは第 1 四分位数の適合度スケーリングした値（$s(list_f.Q1)$），四角プロットは中央値（第 2 四分位数）の適合度スケーリングした値（$s(list_f.Median)$），三角プロットは第 3 四分位数の適合度スケーリングした値（$s(list_f.Q3)$）をそれぞれ表す．なお，適合度スケーリングにおいて f_{max} は 100 として算出している．図 7.21 において横軸は同じく経過単位時間を表し，縦軸は P^{α}_{down} の値を表す．図 7.20 から，進化の過程を通して第 1 四分位数は $f_{max} = 100$ に近い値にスケーリングされており，中央値も進化の前半では $80 = 0.8 \times f_{max}$ 以上の値

表 7.16　各実験ケースにおける 10^7 単位時間経過後の平均適合度とマン–ホイットニーの U 検定によって得られた P 値（2/2）

		R5	R6	R7	R8
Case1	TAGP$^+$	$\mathbf{1.1 \times 10^{-3}}$	$\mathbf{6.2 \times 10^{-4}}$	$\mathbf{7.1 \times 10^{-4}}$	$\mathbf{2.3 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	3.0×10^{-3}	2.9×10^{-2}	7.5×10^{-3}	2.0×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	1.3×10^{-3}	4.1×10^{-3}	1.2×10^{-3}	7.4×10^{-3}
	（P 値）	(0.87)	(0.90)	(0.077)	**(< 0.01)**
Case2	TAGP$^+$	6.6×10^{-4}	$\mathbf{2.2 \times 10^{-3}}$	1.7×10^{-3}	$\mathbf{3.8 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	5.2×10^{-3}	2.7×10^{-2}	1.3×10^{-2}	2.3×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	$\mathbf{5.0 \times 10^{-4}}$	3.6×10^{-3}	$\mathbf{1.0 \times 10^{-3}}$	1.1×10^{-2}
	（P 値）	(0.52)	(0.99)	(0.31)	**(< 0.01)**
Case3	TAGP$^+$	$\mathbf{3.4 \times 10^{-4}}$	6.7×10^{-3}	$\mathbf{1.6 \times 10^{-3}}$	$\mathbf{3.5 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	2.4×10^{-3}	4.8×10^{-2}	5.6×10^{-3}	1.6×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	5.9×10^{-4}	$\mathbf{3.8 \times 10^{-3}}$	2.9×10^{-3}	1.1×10^{-2}
	（P 値）	(0.77)	(0.79)	(0.19)	**(< 0.01)**
Case4	TAGP$^+$	3.5×10^{-4}	$\mathbf{5.2 \times 10^{-4}}$	$\mathbf{2.2 \times 10^{-3}}$	$\mathbf{2.4 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	6.1×10^{-3}	4.9×10^{-2}	2.3×10^{-2}	3.2×10^{-2}
	（P 値）	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	$\mathbf{3.0 \times 10^{-4}}$	6.4×10^{-3}	3.2×10^{-3}	8.6×10^{-3}
	（P 値）	**(0.043)**	(0.072)	**(< 0.01)**	**(< 0.01)**

にスケーリングされていることから集団内の最小適合度に近い個体が半数近く含まれており，多様性が少ないことがわかる．一方，図 7.21 の推移を見ると，集団の多様性が少ないにもかかわらず P^{α}_{down} は高い値で推移しており，「集団内の多様性を保つためにリーパーキュー内の解への移動量を減少」するという設計意図が反映されていないことがわかる．これは，P^{α}_{down} を決定する式 (4.4) において $list_f$ 内の最小値（$list_f.Min$）と同値の要素の個数 n を用いているため，同値ではなく適合度の非常に近い個体で集団内が満たされる今回のケースでは多様性が小さいにもかかわらず n の値が小さくなり，P^{α}_{down} が適切に調節されないことが原因であると考えられる．

そこで，類似した適合度を持つ個体が含まれる場合を考慮して修正を加えた次式の修正

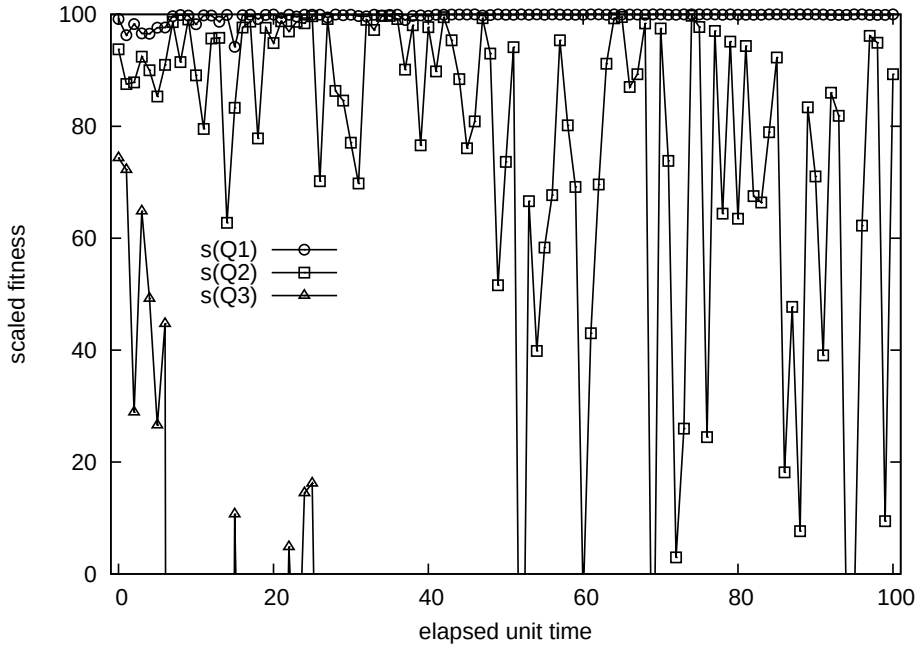


図 7.20　Case2 の R1 における適合度の値が悪い 1 試行の四分位数を適合度スケーリングした値の推移

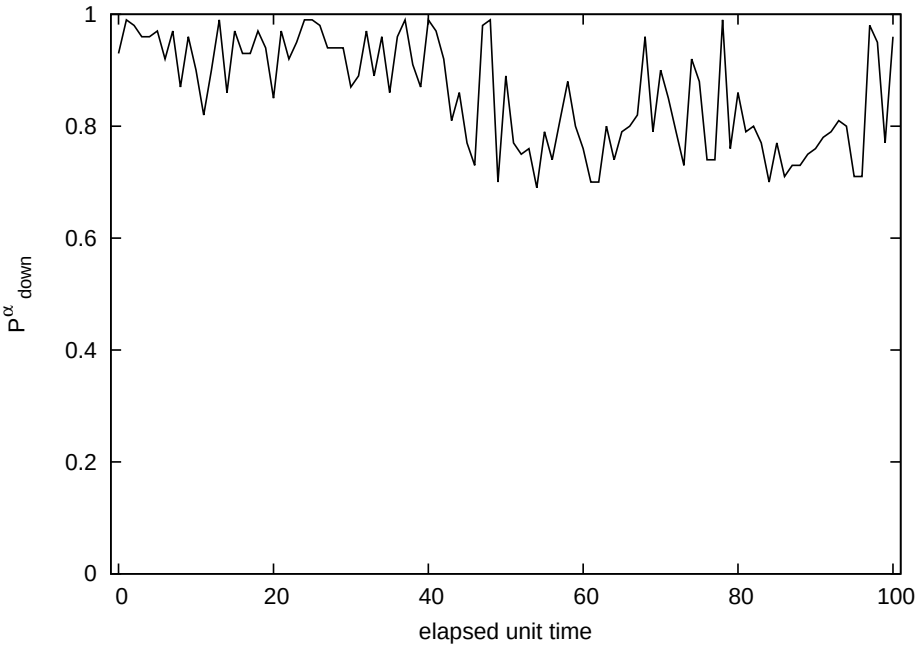


図 7.21　Case2 の R1 における適合度の値が悪い 1 試行のリーパー制御パラメータ P^{α}_{down} の推移

版リーパー制御パラメータ $P^{\alpha*}_{down}$ を用いる実験を行う．

$$P^{\alpha*}_{down} = 1.0 - \frac{\sum_{i=1}^{|list_f|} \left(\frac{s(list_f[i])}{f_{max}} \right)^2}{|list_f|} \tag{7.8}$$

式 (7.8) において，$s(f)$ は適合度スケーリング関数，$list_f[i]$ は $list_f$ の i 番目の要素を表す．$P^{\alpha*}_{down}$ では，$list_f$ 内の最小値と同値の要素数ではなく，スケーリングされた適合度の和を用いることで，適合度の差が小さい個体が多数存在する場合にはその個体を考慮した調節を可能にし，適合度の差が大きい場合には P^{α}_{down} と同等の調整が可能になる．

Case1 から Case4 の R1 における実験結果を図 7.22，7.23，7.24，7.25 に示す．各図は同一経過単位時間における TAGP$^+$，$(\mu+\lambda)$-GP，ASSGP と修正版のリーパー制御パラメータ $P^{\alpha*}_{down}$ を用いる TAGP$^+$（TAGP$^+$ w/ $P^{\alpha*}_{down}$ と表記）の平均適合度の変化を示し，横軸は経過単位時間，縦軸は 30 試行の平均適合度を示す．白抜き四角プロットは TAGP$^+$ の結果，三角プロットは $(\mu+\lambda)$-GP の結果，逆三角プロットは ASSGP の結果を表し，黒塗り四角プロットは TAGP$^+$ w/ $P^{\alpha*}_{down}$ の結果を表す．結果から，すべてのケースにおいて修正版のリーパー制御パラメータを用いる TAGP$^+$ w/ $P^{\alpha*}_{down}$ が他の例題を上回る性能を示すことがわかる．特に，TAGP$^+$ で ASSGP よりも性能の劣る Case2 と Case4 についても結果が改善し，ASSGP を上回っていることが確認できる．TAGP$^+$ w/ $P^{\alpha*}_{down}$ と ASSGP の結果の有意差を統計的に示すために，10^7 単位時間経過後の平均適合度についてマン–ホイットニーの U 検定（Mann–Whitney U test）を用いて検定を行った結果，すべてのケースで有意水準 $\alpha = 0.05$ で有意差があることが確認された．これらの結果から，リーパー制御パラメータを最小値と同値の要素数に基づいた調整ではなく，最小値と差の小さい要素まで考慮することで計算速度にばらつきがある場合でも適切に調整が可能であることが明らかになった．

しかし，評価値との差をどの程度考慮して調整するか，具体的には式 (7.8) の分母の冪数パラメータをどのような値にすべきかが調整度合いに大きな影響を与える．冪数パラメータが小さいほど評価値に大きな差がある場合にもリーパー制御パラメータ $P^{\alpha*}_{down}$ に反映されやすくなるため，類似した適合度を持つ個体が少ない場合でも $P^{\alpha*}_{down}$ は大きくなり，冪数パラメータが 0 の時に $P^{\alpha*}_{down}$ は常に 0 になる．逆に，冪数パラメータが大きいほどより評価値の差の小さな個体のみを考慮した $P^{\alpha*}_{down}$ となり，冪数パラメータが ∞ の時に修正前の P^{α}_{down} と同義になる．このように修正版のリーパー制御パラメータは修正前のパラメータを包括した方法となっており，冪乗パラメータをどのように設定すべきか，また他の例題への影響は今後の課題となる．

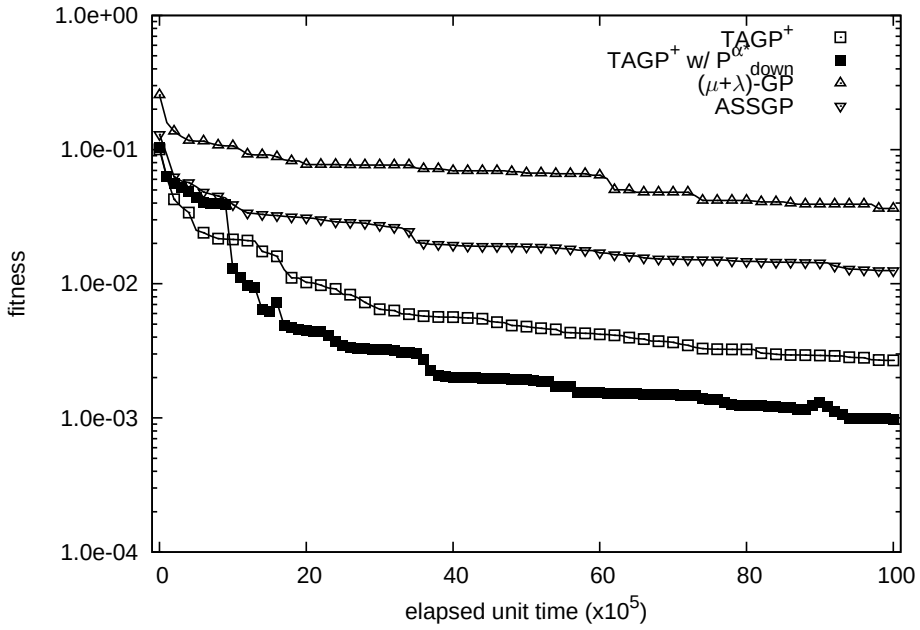


図 7.22　修正版リーパー制御パラメータを用いた TAGP$^+$ の経過単位時間における平均適合度の変化（Case1）

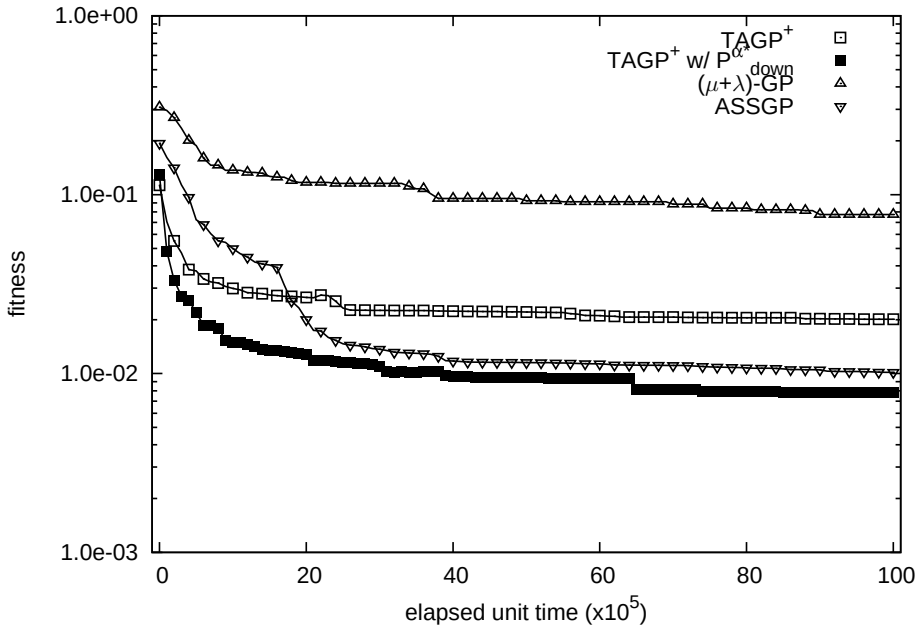


図 7.23　修正版リーパー制御パラメータを用いた TAGP$^+$ の経過単位時間における平均適合度の変化（Case2）

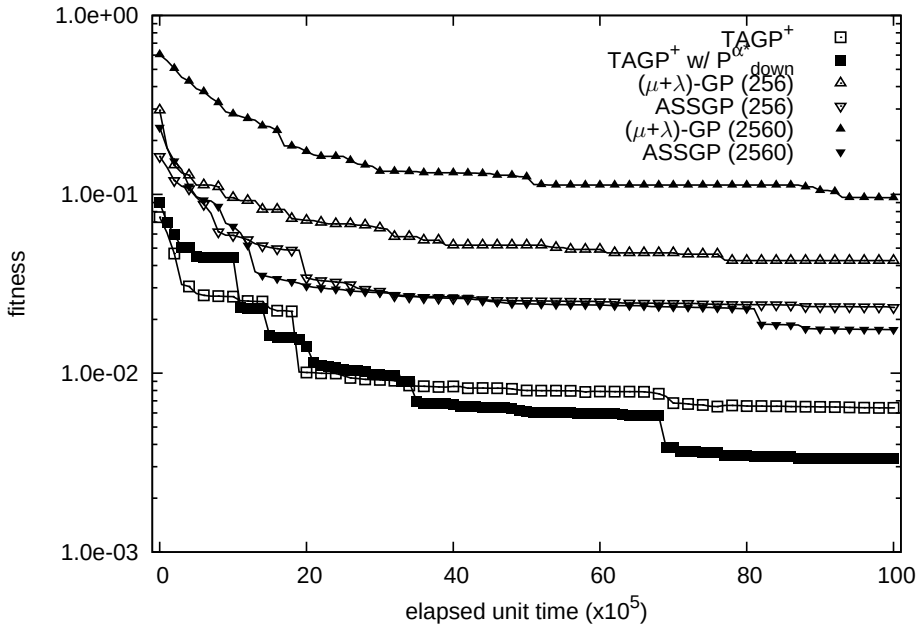


図 7.24　修正版リーパー制御パラメータを用いた TAGP^{+} の経過単位時間における平均適合度の変化（Case3）

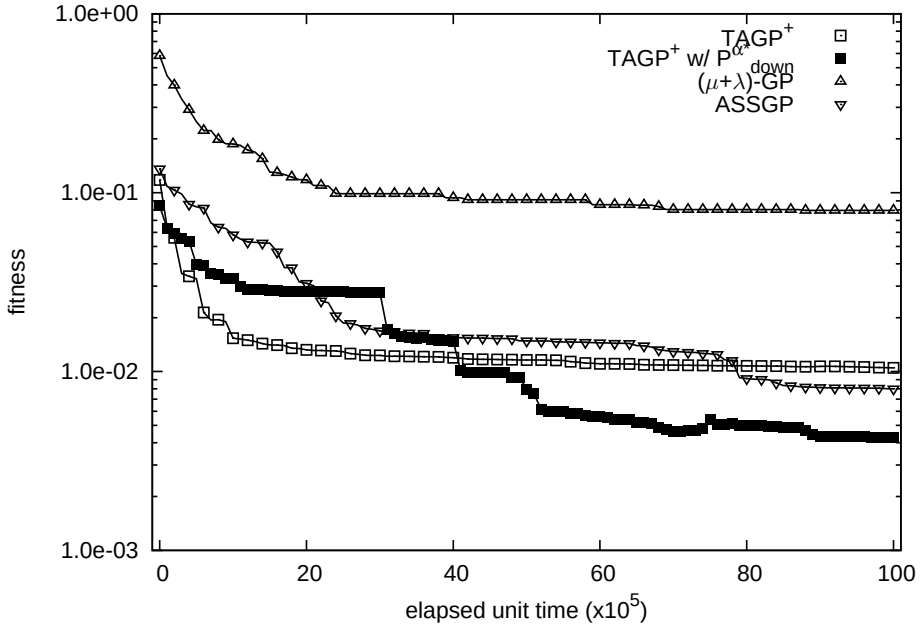


図 7.25　修正版リーパー制御パラメータを用いた TAGP^{+} の経過単位時間における平均適合度の変化（Case4）

第 8 章

実験 2：相対評価を用いる非同期進化的アルゴリズム（ARE-EA）

8.1　概要

本章では，ARE-EA の有効性を検証するために，ARE-EA を遺伝的プログラミング（Genetic Programming：GP）に適用した ARE-GP を用いる実験を行う．まずはじめに 8.2 章で実応用を想定した PIC アセンブリプログラムの進化を用いる実験とその結果を示す．続いて 8.3 章で ARE-EA の宇宙機への適用を念頭においた SEU によるビット反転が起こる環境下で PIC アセンブリプログラムを進化させる実験を行う．最後に 8.4 章で，GP の一般的なベンチマーク問題である関数同定問題を用いる実験とその結果を示す．

8.2　アセンブリプログラム進化

8.2.1　実験内容

本実験では，提案手法である ARE-GP と TAGP^{+}，従来の同期型 GP として steady-state GP（SSGP）[32] を比較する．なお，他の従来手法である $(\mu + \lambda)$-GP と ASSGP に関しては，7.2 章の Case4 の結果からいずれも SSGP と比べて性能が劣ることが確認されたため割愛する．

例題としては，表 8.1 に示す 4 種類の数値計算プログラムと 4 種類の論理演算プログラムの計 8 種類のアセンブリプログラムの進化を扱う．

8.2.2　評価基準と設定

各手法のの共通パラメータ設定を表 8.2 に示す．従来手法である SSGP において，無限ループにより評価が完了しない個体が存在する場合，途中で評価を打ち切る必要があるた

表 8.1　アセンブリプログラム進化の例題（6.2 章より再掲）

数値計算		入力数	出力数	データ数
A1	$f(x) = x^4 + x^3 + x^2 + x$	1	1	16
A2	$f(x) = x^5 - 2x^3 + x$	1	1	16
A3	$f(x) = x^6 - 2x^4 + x^2$	1	1	16
A4	$f(x, y) = x^y$	2	1	25
論理演算		**入力数**	**出力数**	**データ数**
B1	8bit-Parity	8	1	256
B2	7bit-DigitalAdder	7	4	128
B3	6bit-Multiplexer	6	1	64
B4	7bit-Majority	7	1	128

表 8.2　実験パラメータ

最大評価回数	10^6	交叉率	0.7
最大命令数	256	突然変異率	0.1
集団サイズ	100	命令挿入率	0.1
f_{max}	100	命令削除率	0.1

め，その上限を各例題において正常なプログラムが十分実行を完了可能な 50,000 命令とし，上限を超えた個体の適合度は $-\infty$ とする．TAGP$^+$ において，トーナメントサイズ $\lambda = 1$，QFS のパラメータ N_F と N_{update} はいずれも 100 に設定する．ARE-GP において，アーカイブサイズは $\{5, 10, 20, 30\}$ に設定し，アーカイブサイズが 5 のケースについては適合度削除確率 $P_d = \{0.1, 0.3, 0.5, 0.7, 0.9\}$ を実施し，それ以外は $P_d = 0.5$ に設定する．数値計算プログラムでの適合度関数を式 (8.1)，論理演算プログラムでの適合度関数を式 (8.2) にそれぞれ示す．ここで，$\hat{y}$ はプログラムの出力結果，y^* は目標値を表す．

$$f_A = f_{max} - \frac{1}{n}\sum_{i=1}^{n} |\hat{y} - y^*| \tag{8.1}$$

$$f_B = f_{max} \times (1 - \frac{2}{n}\sum_{i=1}^{n} \delta(\hat{y}_i, y_i^*)), \delta(x, y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases} \tag{8.2}$$

個体の優劣は，(1) 適合度，(2) 実行ステップ数，(3) プログラムサイズの順に比較され，適合度は高いほど，実行ステップ数とプログラムサイズは小さいほど優れていると評価する．ARE-GP におけるアーカイブ生成の際の重複判定は適合度，実行ステップ数，プログラムサイズがすべて一致した場合に同一個体であると判定する．

実験は，与えられた目的を達成可能なプログラムで集団を満たした状態から開始する．

各実験は 30 試行ずつ実施し，(1) 目的を達成可能な（*i.e.*, f_{max} の適合度を持つ）プログラムを維持できた割合，(2) 初期プログラムからの実行ステップ数の減少割合の 30 試行平均をもとに評価する．ここで，(1) 目的を達成可能なプログラムを維持できた割合は最大評価回数終了後に集団内に一個体でも目的達成可能なプログラムが存在する場合を維持できたと定義し，(2) 実行ステップ数の減少割合は集団中のプログラムの最小実行ステップ数をもとに評価する．なお，実行ステップ数は各例題についてすべての入出力対を計算し終わるまでに実行した命令数を意味する．また，本実験において (1) 目的を達成可能な（*i.e.*, f_{max} の適合度を持つ）プログラムを維持できた割合は全手法，全例題において 100% であったため，結果としては割愛する．

8.2.3 結果

表 8.3 に従来手法である SSGP，提案手法である TAGP/TDTS（$\lambda = 1$）と ARE-GP の最大評価回数後の実行ステップ数減少割合の 30 試行平均を示す．各例題において最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．なお，適合度削除確率を変化させた結果でアーカイブサイズは 5 に設定しているため，表 8.3 においてアーカイブサイズが 5 の結果と適合度削除確率が 0.5 の結果は同一である．また，SSGP と TAGP$^+$ の結果は 7.2 章の Case4 と同値である．

表 8.3 の結果から，ARE-GP がすべての例題において TAGP$^+$ のような適合度関数の調整なしにリファレンス個体との相対的な比較のみで非同期な進化が可能であることが確認できる．さらに，A4 を除くすべての例題において適切なアーカイブサイズと適合度削除確率を設定することで ARE-GP が SSGP と TAGP$^+$ より優れた結果を示しており，A4 についても TAGP$^+$ とほぼ同等の性能を有していることがわかる．特に，ARE-GP においてアーカイブサイズを 5 と 10 の比較的小さな値に設定した場合にすべての例題で安定して優れた性能を示していることがわかる．これは，アーカイブする際にアーカイブ内の重複を避けるようにしているため，アーカイブサイズが 30 のように大きい場合にアーカイブ内に実行ステップ数の長い個体が残ってしまいリファレンス個体による選択圧が低下してしまうためである．そのため，アーカイブサイズは小さな値に設定することが望ましいと考えられる．

適合度削除確率 P_d の影響に着目すると，$P_d \geq 0.5$ の時にほとんどの問題で TAGP$^+$ を上回る性能を示していることがわかる．一方，数値計算プログラムの例題において P_d が低い場合に TAGP$^+$ よりも性能が劣ることがわかる．実際，$P_d = 0.1$ の場合，初期プログラムからほとんど進化が起こっていないことが確認されている．これは，乗算の計算にループ構造が必要であり，ループ構造を持たない個体が生成された場合，それらの個体の評価時間はループ構造を含む個体に比べ非常に短くなるためである．この時，リーパー削除が実行される間隔が短くなるため，適合度削除確率が低いとループ構造を含む評価時

表 8.3　最終世代の実行ステップ数減少割合の 30 試行平均（1/2）

例題	SSGP	TAGP$^+$	ARE-GP			
			アーカイブサイズ			
			5	10	20	30
A1	59.3% (4.6%)	72.5% (6.6%)	75.5% (3.1%)	75.3% (3.2%)	72.4% (3.4%)	70.7% (3.9%)
A2	58.6% (4.4%)	72.9% (4.9%)	74.5% (3.3%)	74.4% (3.5%)	72.1% (4.3%)	67.5% (4.0%)
A3	56.9% (4.3%)	71.4% (6.6%)	74.6% (3.0%)	72.5% (4.5%)	69.7% (4.9%)	65.4% (4.4%)
A4	73.0% (9.6%)	**76.4%** (13.0%)	75.6% (5.7%)	75.9% (5.3%)	74.6% (9.7%)	75.0% (7.8%)
B1	7.2% (3.6%)	8.3% (3.0%)	8.6% (3.6%)	9.8% (4.4%)	9.1% (5.2%)	5.7% (2.2%)
B2	1.8% (2.3%)	6.3% (3.2%)	7.5% (3.2%)	**8.7%** (4.4%)	7.5% (3.9%)	7.3% (3.3%)
B3	32.8% (8.4%)	54.6% (10.5%)	63.3% (10.4%)	61.7% (10.2%)	60.8% (9.1%)	61.0% (10.5%)
B4	17.9% (16.7%)	26.7% (24.2%)	37.7% (13.9%)	36.4% (13.2%)	**42.7%** (12.3%)	30.8% (12.5%)

間が必要な個体が評価完了前に削除されてしまう．そのため，評価時間が短く適合度の低い個体を積極的に削除し，ループの計算が完了するまでの時間を待機するために，個体間の評価時間差が大きくなる数値計算問題では適合度削除確率を高く設定することが有効であったと考えられる．このことから，ARE-GP では評価時間が短い個体が含まれることを考慮し，適合度削除確率 P_d は大きな値に設定することが適していると言える．

8.2.4　考察 1：計算時間に対する性能の比較

ここでは，7.2 章と同様に計算時間に対する ARE-GP，TAGP$^+$，SSGP，ASSGP の解探索性能の違いを明らかにする．ここでは，並列計算環境において各個体を並列に評価可能な環境を想定し，100 命令実行可能な時間を 1 単位時間とした時の経過単位時間における実行ステップ数の減少割合の 30 試行平均を比較する．表 8.4 に 10^7 単位時間経過後の各手法，各例題の実行ステップ数の平均減少割合を示す．なお，ARE-GP はアーカイブサイズを 5，適合度削除確率を 0.5 にそれぞれ設定した場合の結果を示す．各例題におい

表 8.3　最終世代の実行ステップ数減少割合の 30 試行平均（2/2）

例題	ARE-GP			
	適合度削除確率 P_d			
	0.1	0.3	0.7	0.9
A1	34.6% (14.9%)	74.7% (5.0%)	**76.5%** (2.3%)	75.0% (4.4%)
A2	30.4% (11.8%)	74.5% (3.1%)	75.0% (2.3%)	**75.8%** (1.2%)
A3	23.2% (13.4%)	73.9% (4.8%)	**74.6%** (2.8%)	74.1% (3.7%)
A4	0.0% (0.0%)	0.0% (0.0%)	74.3% (6.8%)	68.7% (13.3%)
B1	6.2% (2.8%)	10.3% (6.0%)	**11.9%** (5.4%)	9.4% (5.1%)
B2	7.5% (2.4%)	8.2% (2.9%)	7.7% (3.0%)	8.5% (3.9%)
B3	**63.6%** (10.5%)	62.5% (8.8%)	62.3% (8.3%)	62.2% (8.1%)
B4	37.7% (19.4%)	28.3% (15.1%)	32.5% (16.4%)	35.8% (17.7%)

て最も減少割合の高い値を太字で示し，括弧の中は標準偏差を表す．表 8.4 において，p 値は ARE-GP と従来手法で最も減少割合の高い ASSGP についてのマン–ホイットニーの U 検定の結果を示す．これらの結果から，ARE-GP が同期 EA である SSGP，および従来の非同期 EA である ASSGP と比べて，評価時間の無駄なく効率的な進化を実現できていることがわかる．実際，マン–ホイットニーの U 検定の結果，すべての例題で有意水準 5% で ARE-GP と ASSGP に有意差が見られることが明らかになった．さらに，TAGP$^+$ と比較しても A4 を除くすべての例題で ARE-GP が高い進化性能を示しているといえる．

以上の結果から，ARE-GP は実行時間の観点から特に同期 EA と従来の非同期 EA を上回る高い進化性能を示し，TAGP$^+$ と同等以上の進化性能を有することが明らかになった．

表 8.4　各手法の 10^7 単位時間経過後の実行ステップ数減少割合の 30 試行平均

例題	SSGP	ASSGP	TAGP$^+$	ARE-GP	p 値
A1	16.4% (2.5%)	37.7% (5.8%)	68.5% (7.7%)	**68.8%** (7.8%)	$< 10^{-3}$
A2	4.0% (16.7%)	35.8% (6.5%)	68.0% (6.4%)	**71.9%** (4.9%)	$< 10^{-3}$
A3	16.5% (3.9%)	35.8% (5.9%)	65.6% (6.2%)	**68.8%** (7.0%)	$< 10^{-3}$
A4	38.0% (2.5%)	52.9% (10.6%)	**73.4%** (9.8%)	70.3% (12.0%)	$< 10^{-3}$
B1	5.5% (2.1%)	7.3% (2.7%)	7.5% (3.3%)	**10.1%** (5.0%)	0.035
B2	0.0% (0.2%)	0.5% (1.1%)	4.1% (6.7%)	**7.0%** (2.3%)	$< 10^{-3}$
B3	16.3% (4.1%)	34.3% (12.0%)	68.3% (10.0%)	**68.8%** (8.6%)	$< 10^{-3}$
B4	1.5% (2.7%)	2.1% (4.9%)	5.0% (9.7%)	**15.2%** (11.0%)	$< 10^{-3}$

8.2.5　考察 2：生成されたプログラム

ARE-GP と TAGP$^+$ の性能の違いを明らかにするために，各手法で例題 A1 と例題 B3 において最終的に獲得されたプログラムを示す．プログラムは各手法 30 試行の中で最も実行ステップ数の短いプログラムを示している．

A1：$f(x) = x^4 + x^3 + x^2 + x$

図 8.1 に例題 A1 で最終的に獲得されたプログラムの一部を示す．図 8.1 に示すプログラムは ARE-GP で得られた最も実行ステップ数の短いプログラムの一部である．図 8.1 では，例題 A1 において $f(x) = x^4 + x^3 + x^2 + x$（$x$ は 0 から 15 の 4 ビットが入力される）の x^4 を計算する部分を示している．図 8.1 のプログラムにおいて，R4，R7 はそれぞれ汎用レジスタを表しており，W はワーキングレジスタを表している．x^4 の計算の際には，R4 レジスタに事前に計算されている x^3 の値が記憶されており，$x^3 \times x$ を実行して x^4 を R4 レジスタに記憶する（最終的には出力用の R0 レジスタに加算される）．“$<-$”は数値の代入を表し，“$>> 1$” は右シフトを表す．

```
 1: BTFSC   R7 0 // if R7[0] == 0 then skip
 2: ADDWF   R4 1 // R4 <- R4 + W
 3: RRF     R4 1 // R4 <- R4 >> 1
 4: RRF     R7 1 // R7 <- R7 >> 1
-------------------------------------------------
... BTFSC   R7 0 // if R7[0] == 0 then skip
... ADDWF   R4 1 // R4 <- R4 + W
... RRF     R4 1 // R4 <- R4 >> 1
... RRF     R7 1 // R7 <- R7 >> 1
    x 7

33: BTFSC   R7 0 // if R7[0] == 0 then skip
34: ADDWF   R4 1 // R4 <- R4 + W
35: RRF     R4 1 // R4 <- R4 >> 1
36: BTFSC   R7 0 // if R7[0] == 0 then skip
37: ADDWF   R4 1 // R4 <- R4 + W
38: RRF     R4 1 // R4 <- R4 >> 1
39: BTFSC   R7 0 // if R7[0] == 0 then skip
40: ADDWF   R4 1 // R4 <- R4 + W
41: RRF     R4 1 // R4 <- R4 >> 1
42: BTFSC   R7 0 // if R7[0] == 0 then skip
43: ADDWF   R4 1 // R4 <- R4 + W
44: RRF     R4 1 // R4 <- R4 >> 1
45: RRF     R4 1 // R4 <- R4 >> 1
46: RRF     R4 1 // R4 <- R4 >> 1
47: RRF     R4 1 // R4 <- R4 >> 1
48: RRF     R4 1 // R4 <- R4 >> 1
49: SWAPF   R4 1 // R4の上位16ビットと下位16ビットを交換
```

図 8.1　ARE-GP によって得られたプログラムの一部（例題 A1）

7.2 章で示したように，TAGP$^+$ では掛け算を計算するためのループを展開することによってループカウンタや GOTO 命令の実行分のステップ数を削減し，さらに SWAPF 命令を用いてレジスタの上位 16 ビットと下位 16 ビットを入れ替えることによって右シフトの実行回数を削減し，結果として実行ステップ数を大幅に削減することができた．ARE-GP で得られたプログラム（図 8.1）においても，同様に x^4 を計算するループを展開し，SWAPF 命令を実行することによってレジスタ値をもとに戻している．さらに，ARE-GP で得られたプログラムは TAGP$^+$ で得られたプログラムからさらに不要な初期化命令等を削除し，TAGP$^+$ で得られたプログラムが 163 命令，2375 ステップであったのに対し，ARE-GP で得られたプログラムは 159 命令，2284 ステップとより簡潔で実行ステップ数の短いプログラムが生成されている．

B3：6bit-Multiplexer

図 8.2，8.3 に例題 B3 で最終的に獲得されたプログラムを示す．図 8.2，8.3 に示すプログラムはどちらも TAGP$^+$，ARE-GP 双方で得られた最も実行ステップ数の短いプログラムであり，図 8.2 は 7.2 章で示したプログラムと同一プログラムである．図 8.2，8.3 のプログラムにおいて，R0 から R7 はそれぞれ汎用レジスタを表しており，W はワーキングレジスタを表している．入力値は R1∼R6 レジスタに入力され，R0 レジスタに結果が出力される．“< −” は数値の代入を表す．

```
 1: SUBWF    R3 0// W <- R3 - W (W = 0)
 2: BTFSC    R2 0// if R2[0] == 0 then skip
 3: MOVF     R4 0// W <- R4
 4: BTFSS    R1 0// if R1[0] == 1 then skip
 5: IORWF    R0 1// R0 <- R0 OR W
 6: MOVF     R5 0// W <- R5
 7: BTFSC    R2 0// if R2[0] == 0 then skip
 8: MOVF     R6 0// W <- R6
 9: BTFSC    R1 0// if R1[0] == 0 then skip
10: IORWF    R0 1// R0 <- R0 OR W
```

図 8.2　TAGP$^+$ によって得られたプログラム（例題 B3）（図 7.11 再掲）

```
 1: MOVF     R3 0// W <- R3
 2: BTFSC    R2 0// if R2[0] == 0 then skip
 3: MOVF     R4 0// W <- R4
 4: BTFSC    R1 0// if R1[0] == 0 then skip
 5: BTFSC    R7 0// if R7[0] == 0 then skip
                    (always true)
 6: IORWF    R0 1// R0 <- R0 OR W
 7: MOVF     R5 0// W <- R5
 8: BTFSC    R2 0// if R2[0] == 0 then skip
 9: MOVF     R6 0// W <- R6
10: BTFSC    R1 0// if R1[0] == 0 then skip
11: IORWF    R0 1// R0 <- R0 OR W
```

図 8.3　ARE-GP によって得られたプログラム（例題 B3）

7.2 章で示したように，TAGP$^+$ で得られたプログラムは初期プログラムで論理演算によって計算していた処理をすべて条件分岐に置き換えることによって実行ステップ数を削減していた．ARE-GP で得られたプログラムも同様に条件分岐によって出力結果を決定することにより実行ステップ数を削減している．しかし，ARE-GP で得られたプログラムは TAGP$^+$ で得られたプログラムとくらべて冗長な条件分岐が一箇所含まれており（図 8.3 中 4，5 行目），TAGP$^+$ で得られたプログラムよりも実行ステップ数が長くなっていることがわかる．この結果，ARE-GP で得られるプログラムは TAGP$^+$ で得られるプ

表 8.5　ビット反転下でのアセンブリプログラム進化の例題（6.3 章より再掲）

数値計算		入力数	出力数	データ数
A1	$f(x) = x^4 + x^3 + x^2 + x$	1	1	16
A2	$f(x) = x^5 - 2x^3 + x$	1	1	16
A3	$f(x) = x^6 - 2x^4 + x^2$	1	1	16
A4	$f(x, y) = x^y$	2	1	25
パリティチェック		入力数	出力数	データ数
E5	5bit-Parity	5	1	32
E6	6bit-Parity	6	1	64
E7	7bit-Parity	7	1	128
E8	8bit-Parity	8	1	256

ログラムよりも余分な命令が 1 命令多く，実行ステップも TAGP$^+$ が 512 ステップであるのに対し，ARE-GP が 544 ステップと ARE-GP がやや劣ることが確認された．

8.3　ビット反転環境下でのアセンブリプログラム進化

8.3.1　実験内容

本実験では，SEU によるビット反転が生じる環境において ARE-EA を GP に適用した ARE-GP が PIC アセンブリプログラムを進化させられるかどうかを検証する．

例題としては，表 8.5 に示す 4 種類の数値計算プログラムと 4 種類の偶数パリティプログラムの計 8 種類のアセンブリプログラムの進化を扱う．

8.3.2　評価基準と設定

ARE-GP のパラメータを表 8.6 に示す．なお，本実験ではトーナメント選択 $\lambda = 1$ の TDTS を用いる TAGP/TDTS を扱い，リーパーキュー制御のパラメータ P_r（P_{down} および P_{up} の上限）は 0.99 に設定する．また，実応用ではメモリ容量が限られることが想定されるため，本実験では母集団サイズを比較的小さい 20 としている．また，単位時間は 100 命令の実行に要する時間と定義する．SEU は単位時間あたりに $\{10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$ の確率でプログラム，またはレジスタのうち 1 ビットが反転するものとする．適合度関数

表 8.6　パラメータ設定

最大単位時間	2.0×10^8	交叉率	0.7
最大プログラムサイズ	256	突然変異率	0.1
母集団サイズ	20	命令挿入率	0.1
適合度削除確率 P_d	0.9	命令削除率	0.1
アーカイブサイズ as	5		

は数値計算プログラムでは式 (8.3)，論理演算プログラムでは式 (8.4) をそれぞれ用いる．

$$f_A = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i^*| \tag{8.3}$$

$$f_E = \frac{1}{n}\sum_{i=1}^{n} \delta(\hat{y}_i, y_i^*), \delta(x,y) = \begin{cases} 1 & x = y \\ 0 & x \neq y \end{cases}, \tag{8.4}$$

ここで，$\hat{y}_i$ は i 番目の入力に対するプログラムの出力結果，y_i^* は i 番目の入力に対する目標値，n はデータ数を表す．個体の優劣は，(1) 適合度，(2) プログラムサイズ，(3) 実行ステップ数の順に比較し評価する．アーカイブ生成の際の重複判定は適合度，プログラムサイズ，実行ステップ数がすべてに一致した場合に同一個体であると判定する．

実験は，与えられた目的を達成可能なプログラムで集団を満たした状態から開始する．各実験は 30 試行ずつ実施し，(1) 目的を達成可能な（*i.e.*, f_{max} の適合度を持つ）プログラムを維持できた割合，(2) 初期プログラムからのプログラムサイズの減少割合の 30 試行平均，(3) 初期プログラムからの実行ステップ数の減少割合の 30 試行平均をもとに評価する．ここで，(1) 目的を達成可能なプログラムを維持できた割合は最大評価回数終了後に集団内に一個体でも目的達成可能なプログラムが存在する場合を維持できたと定義し，(2) プログラムサイズの減少割合と (3) 実行ステップ数の減少割合は集団中のプログラムの最小実行ステップ数をもとに評価し，目的を達成可能なプログラムを維持できた試行における平均とする．また，実行ステップ数は各例題についてすべての入出力対を計算し終わるまでに実行した命令数を意味する．

8.3.3　結果

表 8.7 に (1) 目的を達成可能なプログラムを維持できた割合，表 8.8 に (2) プログラムサイズの減少割合の 30 試行平均，表 8.9 に (3) 実行ステップ数の減少割合の 30 試行平均をそれぞれ示す．なお，表 8.8，8.9 において，負値は減少率，正値は増加率を表す．まず，表 8.7 の結果から，ビット反転が 10^{-4} よりも低い場合にはすべての例題で目的を達成可能なプログラムを維持できていることがわかる．一方，ビット反転率が 10^{-3} と極めて高い場合，特にループを含む数値計算プログラムの例題で正常なプログラムを維持でき

表 8.7　(1) 目的を達成可能なプログラムを維持できた割合

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
A1	0.8	1.0	1.0	1.0
A2	0.53	1.0	1.0	1.0
A3	0.53	1.0	1.0	1.0
A4	1.0	1.0	1.0	1.0
E5	1.0	1.0	1.0	1.0
E6	1.0	1.0	1.0	1.0
E7	1.0	1.0	1.0	1.0
E8	0.93	1.0	1.0	1.0

表 8.8　(2) プログラムサイズの減少割合の 30 試行平均

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
A1	−15.7%	−29.1%	−29.7%	−29.6%
A2	−11.2%	−24.7%	−25.1%	−24.9%
A3	−15.3%	−27.9%	−27.9%	−28.0%
A4	−29.5%	−30.7%	−33.5%	−28.9%
E5	−28.0%	−29.3%	−31.6%	−27.6%
E6	−17.5%	−14.9%	−16.3%	−20.0%
E7	−13.5%	−10.7%	−9.5%	−8.8%
E8	−8.3%	−6.0%	−5.2%	−6.3%

表 8.9　(3) 実行ステップ数の減少割合の 30 試行平均

	10^{-3}	10^{-4}	10^{-5}	10^{-6}
A1	24.9%	1.4%	2.1%	−4.7%
A2	19.3%	13.4%	−0.5%	−0.5%
A3	20.6%	1.9%	3.0%	−1.3%
A4	4.4%	−28.1%	−13.5%	−3.8%
E5	−13.0%	−16.8%	−21.0%	−19.3%
E6	−0.2%	1.9%	1.2%	−5.7%
E7	7.8%	2.5%	0.8%	6.6%
E8	17.8%	16.9%	14.8%	7.8%

ていないことがわかる．また，7.3 章で示した TAGP を用いた結果と比較してビット反転率が 10^{-3} の場合にプログラムを維持できる割合が低下していることがわかる．これは，ARE-GP では優良個体を保持するアーカイブを使用するが，アーカイブに対してビット反転が生じる場合に適切な進化が継続できないためである．次に，表 8.8 の結果から，すべての例題において進化によってプログラムサイズを減少できていることがわかる．特に，7.3 章で示した TAGP を用いた場合と比べても高いプログラムサイズ減少率を示しており，さらに表 8.9 の結果から実行ステップ数の減少率も TAGP を上回っていることから，ARE-GP が TAGP と比較して SEU によるビット反転が生じる環境でも高い進化性能を有していることがわかる．以上の結果から，ARE-GP は SEU によるビット反転が生じる環境において TAGP と比較して高ビット反転率に対する頑健性は劣るものの，目的を達成可能なプログラムを維持可能であり，さらに TAGP よりも高い進化性能を有することが明らかになった．

8.3.4 考察

進化によって実際に獲得されたプログラムの例を図 8.4，8.5 に示す．図 8.4 は，例題 A1（$f(x) = x^4 + x^3 + x^2 + x$）における初期プログラムと進化後のプログラム，図 8.5 は，例題 E5（5bit-Parity）における初期プログラムと進化後のプログラムをそれぞれ表す．

まず，例題 A1 において，初期プログラムでは x^2，x^3，x^4 の計算をそれぞれ 3 つのループ（初期プログラム 9 行目から 17 行目，27 行目から 35 行目，45 行目から 53 行目）で実行し R2，R3，R4 レジスタに記憶している．そして，最後に ADDWF 命令を 4 回実行し，R1（入力値 x），R2，R3，R4 の値を順に出力用のレジスタである R0 に加算している．それに対し，進化後のプログラムでは，交叉によって初期プログラムにおける 2 番目のループ（初期プログラム 27 行目から 35 行目）が 3 番目のループの箇所（進化後プログラム 33 行目から 40 行目）にコピーされ，さらに 3 番目のループの前に ADDWF 命令が加わっている（進化後プログラム 27 行目）．これにより，R3 レジスタと W レジスタに $x^3 + x$ の値が記憶された状態になり，さらに 3 番目のループで $\left(x^3 + x\right) \times x = x^4 + x^2$ が計算され，順次 R3 レジスタに加算されている．そして，最終的に ADDWF 命令によって R3 レジスタの値が R0 レジスタに代入され計算が完了する．この変更により，進化前には 3 つのレジスタ（R2，R3，R4）に別々に計算結果を格納し最後に加算する処理が不要になり，その分のプログラムサイズを削減されている．

次に，例題 E5 において，初期プログラムでは R6 レジスタに入力値である R1 から R5 までのレジスタ値を加算し（初期プログラム 2 行目から 11 行目），最後に R6 レジスタの 0 ビット目の値に基づいて出力値を決定している．この時，PIC マイコンの命令では汎用レジスタから汎用レジスタへの代入命令が存在しないため，W レジスタを一度経由してから計算をしている．それに対し，進化後のプログラムでは，ほとんどの部分を R6 レジ

初期プログラム

```
 1: CLRF     R2 1      // R2 <- R1 x R1
 2: MOVF     R1 0      // W <- R1
 3: MOVWF    R5 1      // R5 <- W
 4: MOVWF    R6 1      // R6 <- W
 5: MOVF     R6 0      // W <- R6
 6: MOVWF    R7 1      // R7 <- W
 7: MOVLW    32        // W <- 32
 8: MOVWF    R6 1      // R6 <- 8
 9: LABEL0
10: MOVF     R5 0      // W <- R5
11: BCF      STATUS 0
12: BTFSC    R7 0      // if R7[0] == 0 then skip
13: ADDWF    R2 1      // R2 <- R2 + W
14: RRF      R2 1      // R2 <- R2 >> 1
15: RRF      R7 1      // R7 <- R7 >> 1
16: DECFSZ   R6 1      // R6 <- R6 - 1
                          and if R6 == 0 then skip
17: GOTO     LABEL0
18: CLRF     R3 1      // R3 <- R2 x R1
19: MOVF     R1 0      // W <- R1
20: MOVWF    R5 1      // R5 <- W
21: MOVF     R2 0      // W <- R2
22: MOVWF    R6 1      // R6 <- W
23: MOVF     R6 0      // W <- R6
24: MOVWF    R7 1      // R7 <- W
25: MOVLW    32        // W <- 32
26: MOVWF    R6 1      // R6 <- 8
27: LABEL1
28: MOVF     R5 0      // W <- R5
29: BCF      STATUS 0
30: BTFSC    R7 0      // if R7[0] == 0 then skip
31: ADDWF    R3 1      // R3 <- R3 + W
32: RRF      R3 1      // R3 <- R3 >> 1
33: RRF      R7 1      // R7 <- R7 >> 1
34: DECFSZ   R6 1      // R6 <- R6 - 1
                          and if R6 == 0 then skip
35: GOTO     LABEL1
36: CLRF     R4 1      // R4 <- R3 x R1
37: MOVF     R1 0      // W <- R1
38: MOVWF    R5 1      // R5 <- W
39: MOVF     R3 0      // W <- R3
40: MOVWF    R6 1      // R6 <- W
41: MOVF     R6 0      // W <- R6
42: MOVWF    R7 1      // R7 <- W
43: MOVLW    32        // W <- 32
44: MOVWF    R6 1      // R6 <- 8
45: LABEL2
46: MOVF     R5 0      // W <- R5
47: BCF      STATUS 0
48: BTFSC    R7 0      // if R7[0] == 0 then skip
49: ADDWF    R4 1      // R4 <- R4 + W
50: RRF      R4 1      // R4 <- R4 >> 1
51: RRF      R7 1      // R7 <- R7 >> 1
52: DECFSZ   R6 1      // R6 <- R6 - 1
                          and if R6 == 0 then skip
53: GOTO     LABEL2
54: CLRF     R0 1      // R0 <- 0
55: MOVF     R1 0      // W <- R1
56: ADDWF    R0 1      // R0 <- R0 + W
57: MOVF     R2 0      // W <- R2
58: ADDWF    R0 1      // R0 <- R0 + W
59: MOVF     R3 0      // W <- R3
60: ADDWF    R0 1      // R0 <- R0 + W
61: MOVF     R4 0      // W <- R4
62: ADDWF    R0 1      // R0 <- R0 + W
```

図 8.4　例題 A1（$f(x) = x^4 + x^3 + x^2 + x$）における初期プログラムと進化後のプログラム（1/2）

進化後プログラム

```
 1: MOVF     R1 0     // W <- R1
 2: XORWF    R5 1     // R5 <- R5 XOR W (R5 <- W)
 3: IORWF    R7 1     // R7 <- R7 IOR W (R7 <- W)
 4: MOVLW    32       // W <- 32
 5: MOVWF    R6 1     // R6 <- W
 6: LABEL0
 7: MOVF     R5 0     // W <- R5
 8: BTFSC    R7 0     // if R7[0] == 0 then skip
 9: ADDWF    R2 1     // R2 <- R2 + W
10: RRF      R2 1     // R2 <- R2 >> 1
11: RRF      R7 1     // R7 <- R7 >> 1
12: DECFSZ   R6 1     // R6 <- R6 - 1
                         and if R6 == 0 then skip
13: GOTO     LABEL0
14: MOVF     R2 0     // W <- R2
15: MOVWF    R7 1     // R7 <- W
16: MOVLW    32       // W <- 32
17: MOVWF    R6 1     // R6 <- W
18: LABEL1
19: MOVF     R5 0     // W <- R5
20: BTFSC    R7 0     // if R7[0] == 0 then skip
21: ADDWF    R3 1     // R3 <- R3 + W
22: RRF      R3 1     // R3 <- R3 >> 1
23: RRF      R7 1     // R7 <- R7 >> 1
24: DECFSZ   R6 1     // R6 <- R6 - 1
                         and if R6 == 0 then skip
25: GOTO     LABEL1
26: MOVF     R5 0     // W <- R5
27: ADDWF    R3 1     // R3 <- R3 + W
28: MOVLW    32       // W <- 32
29: MOVWF    R6 1     // R6 <- W
30: MOVLW    214      // W <- 214
31: MOVF     R3 0     // W <- R3
32: MOVWF    R7 1     // R7 <- W
33: LABEL1
34: MOVF     R5 0     // W <- R5
35: BTFSC    R7 0     // if R7[0] == 0 then skip
36: ADDWF    R3 1     // R3 <- R3 + W
37: RRF      R3 1     // R3 <- R3 >> 1
38: RRF      R7 1     // R7 <- R7 >> 1
39: DECFSZ   R6 1     // R6 <- R6 - 1
                         and if R6 == 0 then skip
40: GOTO     LABEL1
41: MOVF     R3 0     // W <- R3
42: ADDWF    R0 1     // R0 <- R0 + W
```

図 8.4　例題 A1（$f(x) = x^4 + x^3 + x^2 + x$）における初期プログラムと進化後のプログラム（2/2）

スタを用いずに直接 W レジスタを用いて計算している（進化後プログラム 1 行目から 7 行目）．これにより，初期プログラムから W レジスタを経由した代入を実行する分の命令が削減されており，初期よりも小さなプログラムサイズが実現されている．

8.4　関数同定問題

8.4.1　実験内容

本実験では，提案手法である ARE-GP と TAGP$^+$，同期進化型 GP として個体の評価を並列に実行することで容易に並列化が可能な $(\mu + \lambda)$ 選択を用いる GP（以下，

初期プログラム

```
 1: CLRF     R6 1     // R6 <- 0
 2: MOVF     R1 0     // W <- R1
 3: ADDWF    R6 1     // R6 <- R6+W
 4: MOVF     R2 0     // W <- R2
 5: ADDWF    R6 1     // R6 <- R6+W
 6: MOVF     R3 0     // W <- R3
 7: ADDWF    R6 1     // R6 <- R6+W
 8: MOVF     R4 0     // W <- R4
 9: ADDWF    R6 1     // R6 <- R6+W
10: MOVF     R5 0     // W <- R5
11: ADDWF    R6 1     // R6 <- R6+W
12: MOVLW    0        // W <- 0
13: BTFSC    R6 0     // if R6 & 0x0001
14: MOVLW    1        // W <- 1
15: MOVWF    R0  1    // R0 <- W
```

進化後プログラム

```
1: MOVF     R1 0     // W <- R1
2: SUBWF    R3 0     // W <- R3 - W
3: ADDWF    R2 0     // W <- R2 + W
4: ADDWF    R5 0     // W <- R5 + W
5: ADDWF    R6 1     // R6 <- R6 + W
6: MOVF     R4 0     // W <- R4
7: ADDWF    R6 1     // R6 <- R6 + W
8: BTFSC    R6 0     // if R6[0] == 0 then skip
9: INCFSZ   R0 1     // R0 <- R0 + 1
```

図 8.5　例題 E5（5bit-Parity）における初期プログラムと進化後のプログラム

表 8.10　関数同定問題の例題（6.1 章より再掲）

R1	$f(x) = x^4 + x^3 + x^2 + x$
R2	$f(x) = x^5 - 2x^3 + x$
R3	$f(x) = x^6 - 2x^4 + x^2$
R4	$f(x, y) = x^y$
R5	$f(x) = sin(x^2) \times cos(x) - 1$
R6	$f(x) = sin(x) + sin(x + x^2)$
R7	$f(x) = ln(x + 1) + ln(x^2 + 1)$
R8	$f(x, y) = sin(x) + sin(y^2)$

$(\mu + \lambda)$-GP)，従来法である非同期 steady-state GP[17]（以下，ASSGP）を比較する．

例題としては，表 8.10 に示す 8 種類の関数を扱う．

本実験では，並列計算環境において，(**Case1**) 全個体が同一の評価速度を持つ場合，(**Case2**) 各個体の計算速度にばらつきがある場合，(**Case3**) 評価が完了しない（評価に失敗する）個体が含まれる場合，および (**Case4**)Case2 と Case3 が同時に起こる場合を想定し，ARE-GP と $(\mu + \lambda)$-GP を比較する．以下，それぞれの環境設定について説明する．

表 8.11　Case2 における単位時間あたりの実行可能命令のばらつき

単位時間あたりの実行可能命令数	個体数
20	3
40	3
60	20
80	43
100	31

Case1：全個体が同一の評価速度を持つ場合

Case1 では，全個体が単位時間あたりに同一の命令数（100 命令）を実行可能であり，かつ全個体は必ず評価が完了する．ここで，「単位時間あたりの実行可能命令数」とは 1 単位時間あたりに実行可能な命令数を表す．例えば，単位時間あたりの実行可能命令数が 100 命令であり，命令数（プログラムサイズ）が 100 命令の個体の場合には 1 単位時間ですべての命令を実行可能である．例題のデータ数が 100 個であることから 100 単位時間で 1 回の評価が完了できることを意味する．なお，本実験では 100 個体の並列評価を想定し，$(\mu + \lambda)$-GP では 1 世代進むごとにその世代における最大経過単位時間が経過，ARE-GP ではリーパー削除による評価の打ち切りも含めた並列の評価時間が経過するものとする．

Case2：各個体の計算速度にばらつきがある場合

Case1 では，全個体が一律に単位時間あたりに 100 命令を実行できるのに対し，Case2 では各個体の計算速度にばらつきを持たせる．具体的には，表 8.11 に示す割合で 100 個体中 3 個体は単位時間あたり 20 命令，3 個体は 40 命令，20 個体は 60 命令，43 個体は 80 命令，残りの 31 個体は 100 命令をそれぞれ実行できるものとする．この設定により，最も評価時間の短い個体（単位時間あたり 100 命令）に比べ，最も評価時間の長い個体（単位時間あたり 20 命令）は 5 倍の時間が必要になる．この環境は，並列計算環境において，各計算機の計算速度が異なる状況に相当する．

Case3：評価が完了しない個体が含まれる場合

Case1 では，全個体が必ず評価を完了したのに対し，Case3 ではある一定の割合 P_f の個体が評価が完了しない．本実験では，$P_f = 0.05$ に設定し，全評価の 5% が完了しない状況を模擬する．具体的には，進化の過程において常にランダムな 5% の個体のプログラムカウンタが増加しないようにし，プログラムの実行が完了しないようにする．同期進化である $(\mu + \lambda)$-GP では全個体の評価値を算出する必要があるため，評価の待機時間に上

表 8.12　実験パラメータ

最大命令数	256 命令	突然変異率	0.1
集団サイズ	100	命令挿入率	0.1
交叉率	0.7	命令削除率	0.1
交叉法	二点交叉		

限を設け，評価を打ち切る必要がある．本実験では，評価時間が設定した待機単位時間（詳細は 8.4.3 章）を超過した場合は，その個体の評価値を $-\infty$ に設定する．この環境は，通信エラーや計算機の故障によって計算結果が得られない状況や解自体の性質（例えば，無限ループ）によって計算が終了しない状況に相当する．

Case4：Case2 と Case3 が同時に起こる場合

Case4 では，Case2 と Case3 が同時に起こる場合を想定する．具体的には，単位時間あたりの実行可能命令数は図 8.11 によって決定し，P_f は 0.05 に設定する．

8.4.2　評価基準と設定

適合度関数は式 (8.5) に示す平均二乗誤差を用いる．

$$f_{reg} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i^*)^2 \tag{8.5}$$

ここで，$\hat{y}_i$ は i 番目の入力に対するプログラムの出力結果，y_i^* は i 番目の入力に対する目標値，n はデータ数（$= 100$）を表す．

パラメータの影響分析では，同一評価回数（1.0×10^6 回）における集団内の最小適合度をもとに評価する．

同期 GP との比較では，100 台の並列計算機で並列に評価可能な並列計算環境において各個体の評価時間が異なる環境を模擬した実験を行う．本実験では，同一評価回数における比較と，同一経過時間における比較の 2 種類の尺度で ARE-GP と $(\mu+\lambda)$-GP，ASSGP を比較する．評価回数は 1.0×10^6 回，経過時間は 1.0×10^7 単位時間を上限とする．なお，単位時間は並列な時間経過をもとに算出する．実験は各ケース 30 試行ずつ行い，集団内の最小適合度の平均をもとに評価する．

パラメータ設定は ARE-GP，TAGP$^+$，$(\mu+\lambda)$-GP，ASSGP のすべての手法で共通に表 8.12 に示す値を使用する．ARE-EA と $(\mu+\lambda)$-GP，ASSGP を比較する実験で，ARE-EA では適合度削除確率 $P_d = 0.9$，アーカイブサイズ $as = 5$ の設定を使用し，アーカイブ生成の際の重複判定は適合度，プログラムサイズがともに一致した場合に同一個体であると判定する．TAGP$^+$ において，適合度スケーリングと P_{down}^{α} の調整に用いるパ

ラメータ N_f と N_{update} はともに 100 に設定する．

8.4.3 結果

Case1：全個体が同一の評価速度を持つ場合

図 8.6 に Case1 において同一の評価回数における ARE-GP と TAGP$^+$，$(\mu+\lambda)$-GP，ASSGP の平均適合度の変化を示す．図 8.6 において，横軸は評価回数，縦軸は 30 試行の平均適合度を示す．丸プロットは ARE-GP の結果，四角プロットは TAGP$^+$ の結果，三角プロットは $(\mu+\lambda)$-GP の結果，逆三角プロットは ASSGP の結果を表す．図 8.6 の結果から，ARE-GP の方が初期の収束までに多くの評価回数が必要であるが，最終世代においては $(\mu+\lambda)$-GP と同等以上の性能を示しており，ASSGP と比較してもほぼ同等の性能を示していることがわかる．

これに対し，同一の経過単位時間における ARE-GP と TAGP$^+$，$(\mu+\lambda)$-GP，ASSGP の平均適合度の変化を図 8.7 に示す．図 8.7 において横軸は経過単位時間，縦軸は 30 試行の平均適合度を示し，丸プロットが ARE-GP の結果，四角プロットは TAGP$^+$ の結果，三角プロットが $(\mu+\lambda)$-GP の結果，逆三角プロットは ASSGP の結果を表す．

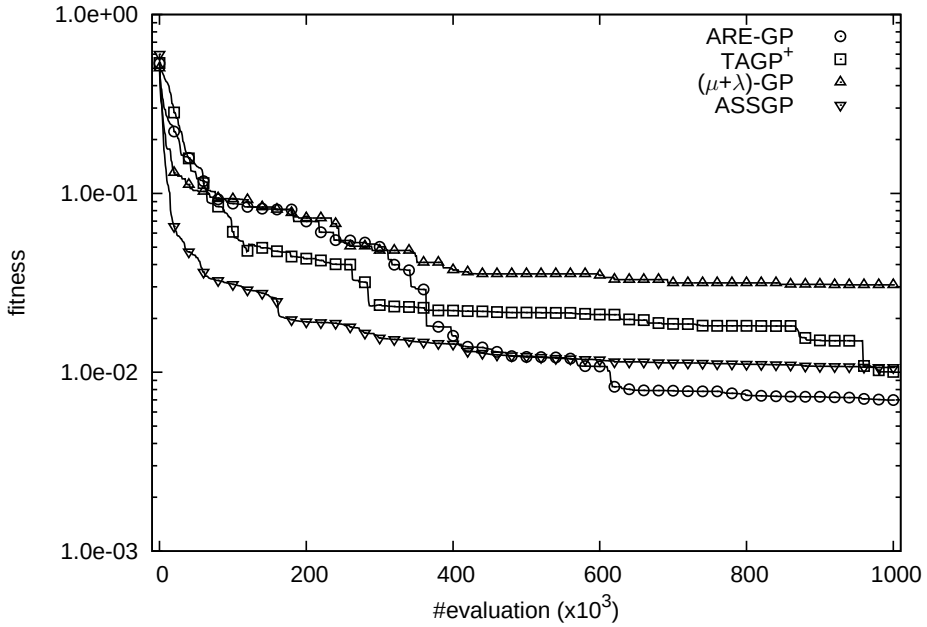
結果から，同一の経過時間で比較した場合，ARE-GP は $(\mu+\lambda)$-GP，ASSGP に比べ，いずれの例題においても収束速度，最終世代での平均的適合度ともに優れていることがわかる．これは，$(\mu+\lambda)$-GP が評価時間の最も遅い個体（本実験では，プログラムサイズの最も大きい個体）の評価が完了してからでないと子個体が生成できないため，待機時間分の評価時間が無駄になるのに対し，ARE-GP は他の個体の評価を待たずに 2 個体の評価が完了した段階で子個体を生成するため，より多く子個体を生成できるためである．

以上の結果から，ARE-GP は同一の評価回数では $(\mu+\lambda)$-GP，ASSGP に比べ収束速度が劣るものの同等の進化を実現でき，さらに単位時間あたりの効率的な子個体生成により，並列計算時に単位時間あたりの解探索性能が同期進化の $(\mu+\lambda)$-GP，および従来の非同期 GP である ASSGP よりも優れているといえる．

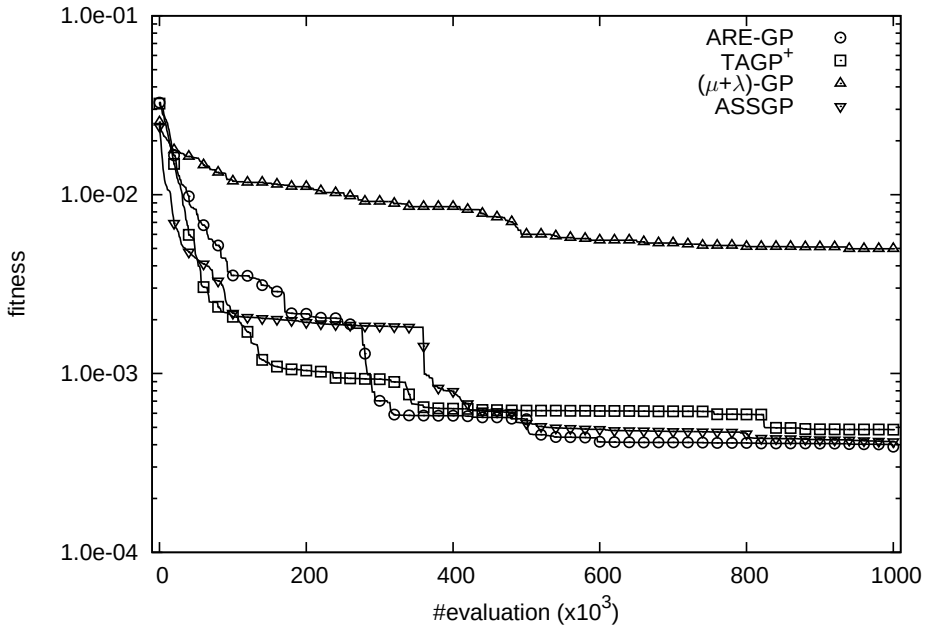
Case2：各個体の計算速度にばらつきがある場合

各個体の計算速度にばらつきのある Case2 の同一の経過単位時間における平均適合度の変化を図 8.8 に示す．図 8.8 において，各軸，各プロットは図 8.7 と同一である．

結果から，ARE-GP は収束速度，最終世代の平均適合度ともに Case1 の結果と同等の結果が得られていることがわかる．また，ARE-GP と ASSGP を比較すると，R3 と R6，R7 では ASSGP が収束速度，最終世代の平均適合度ともに ARE-GP を上回っているものの，それ以外の例題で ARE-GP が ASSGP とくらべて収束速度が上回っており，最終世代の平均適合度も ARE-GP が ASSGP と同等以上の性能を示している．以上の結果から，ARE-GP は各個体の計算速度にばらつきがある場合においても解探索性能を低下さ

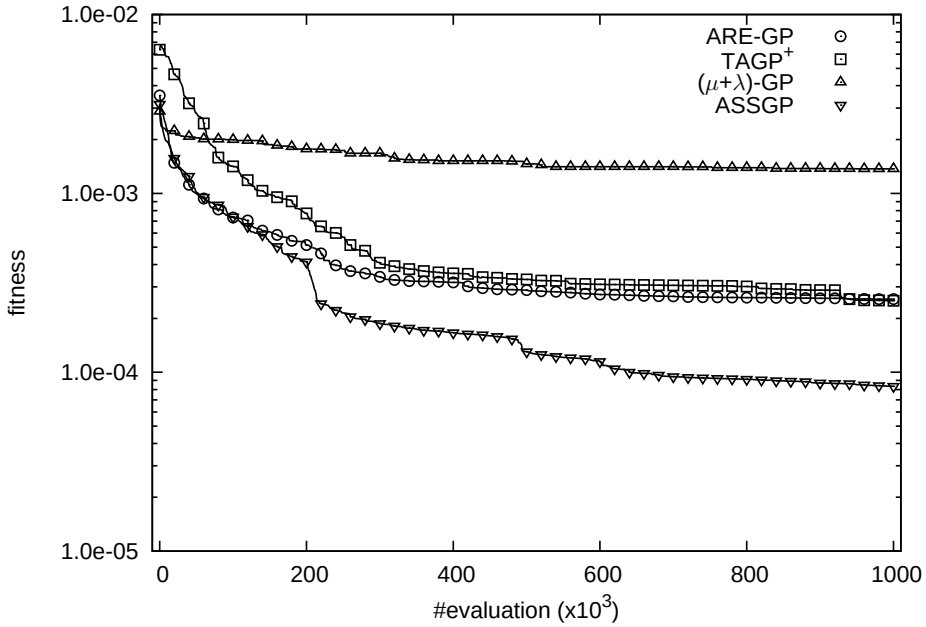


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

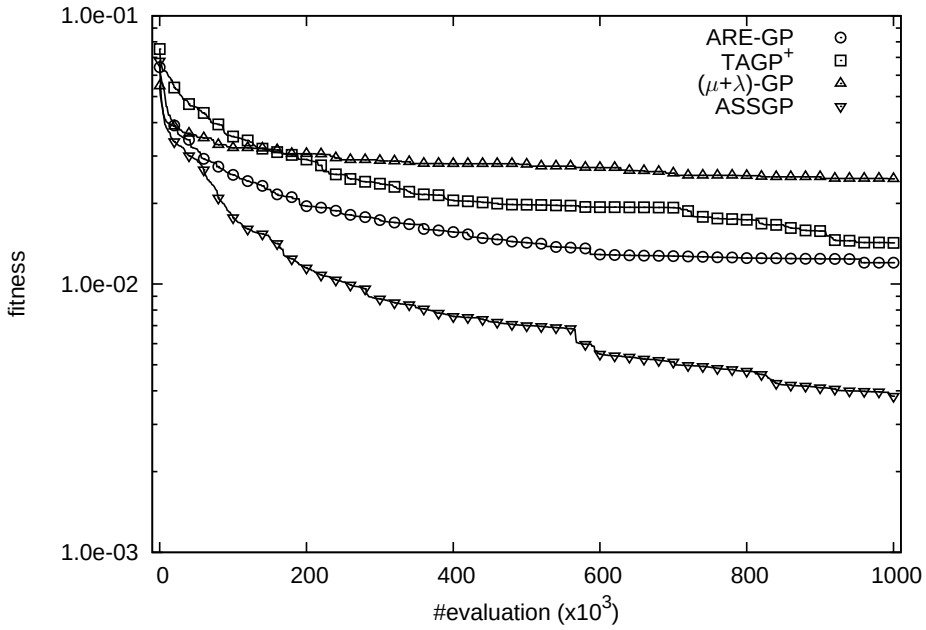


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 8.6　Case1：同一の評価回数における平均適合度の変化（1/4）

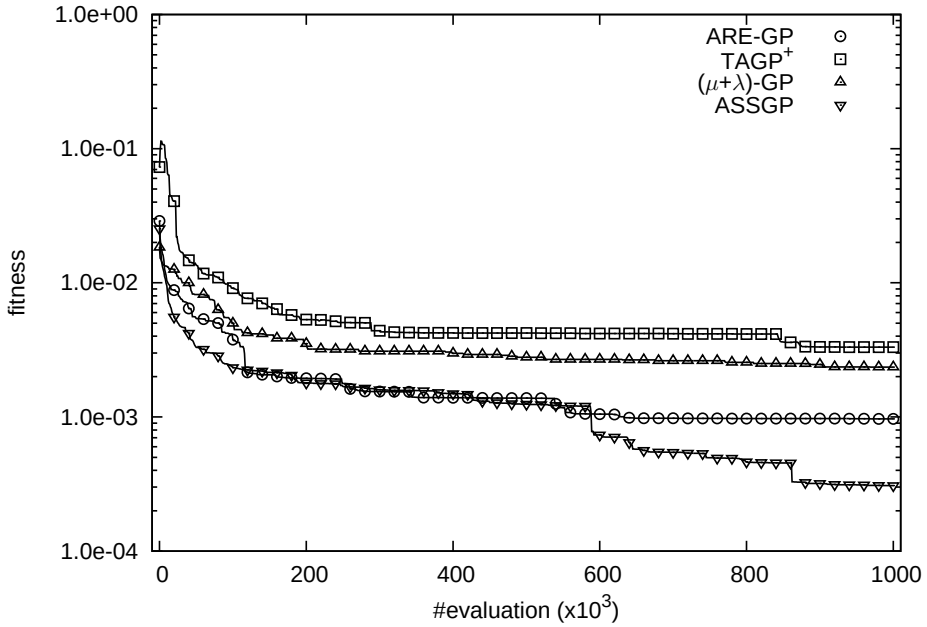


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

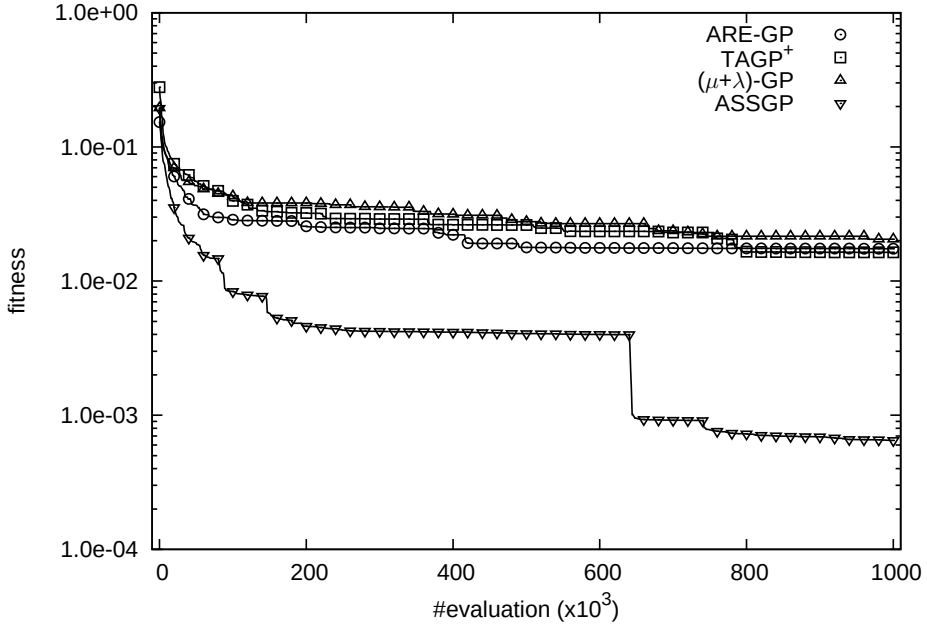


(d) R4（$f(x, y) = x^y$）

図 8.6　Case1：同一の評価回数における平均適合度の変化（2/4）

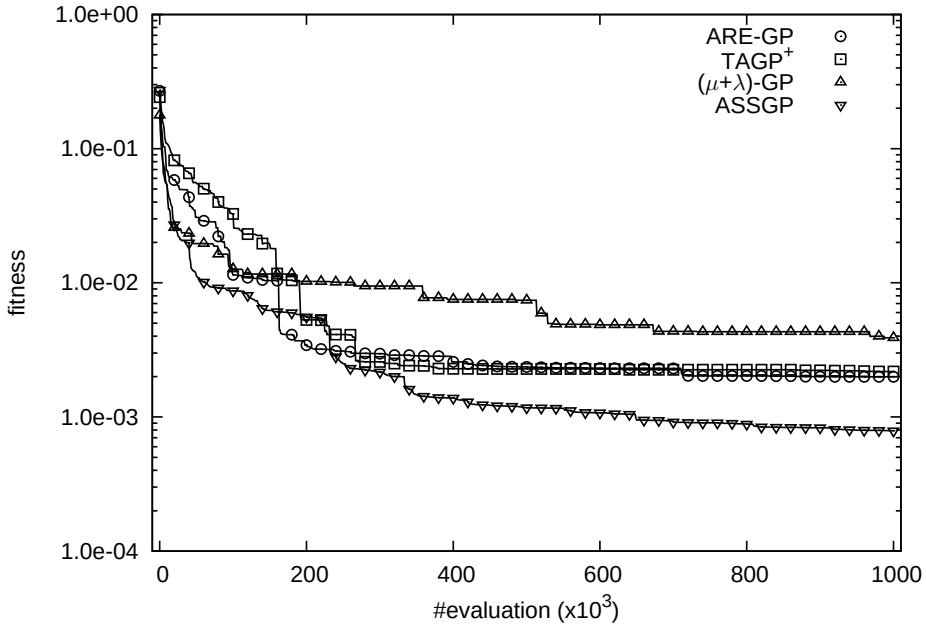


(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

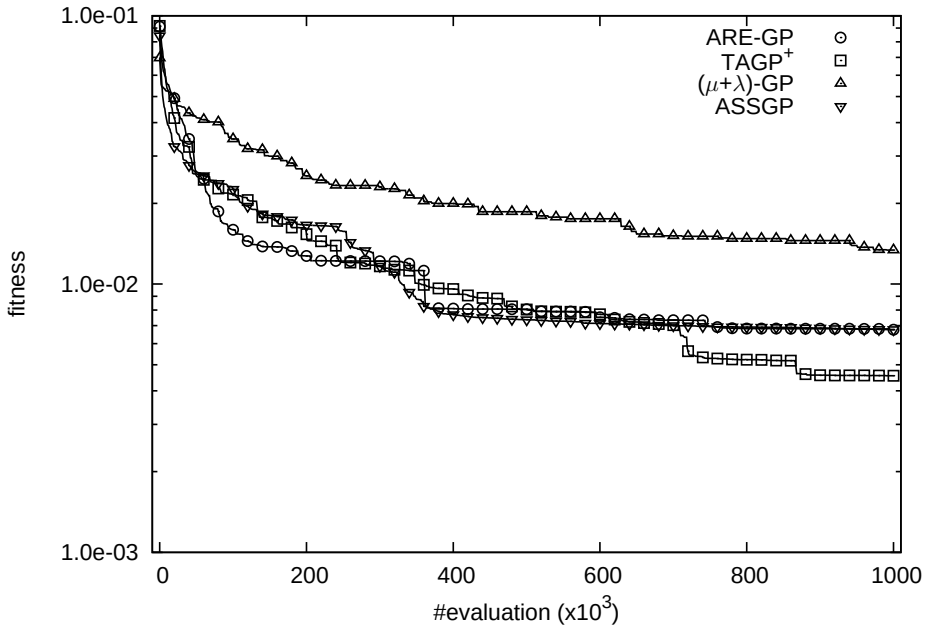


(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 8.6　Case1：同一の評価回数における平均適合度の変化（3/4）

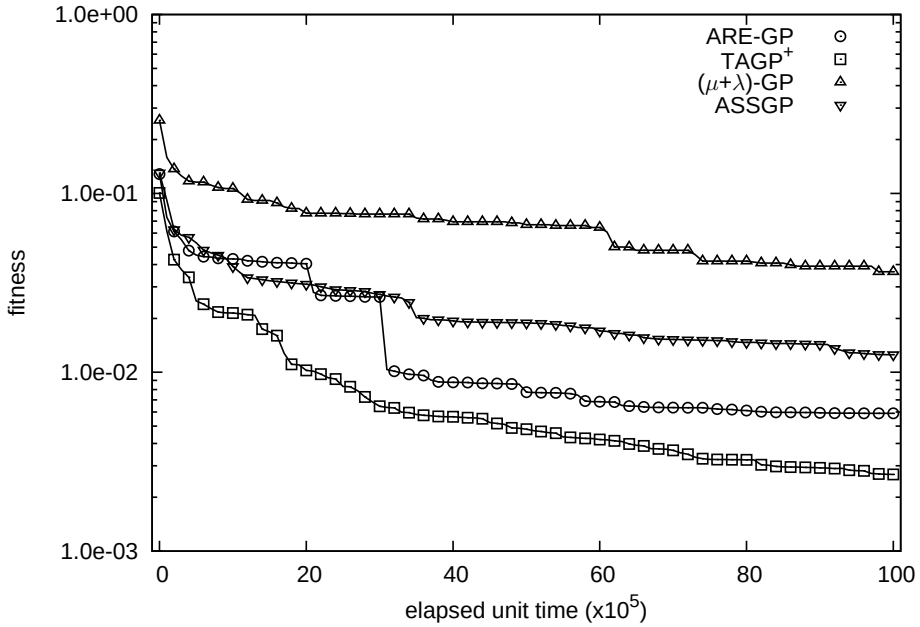


(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）

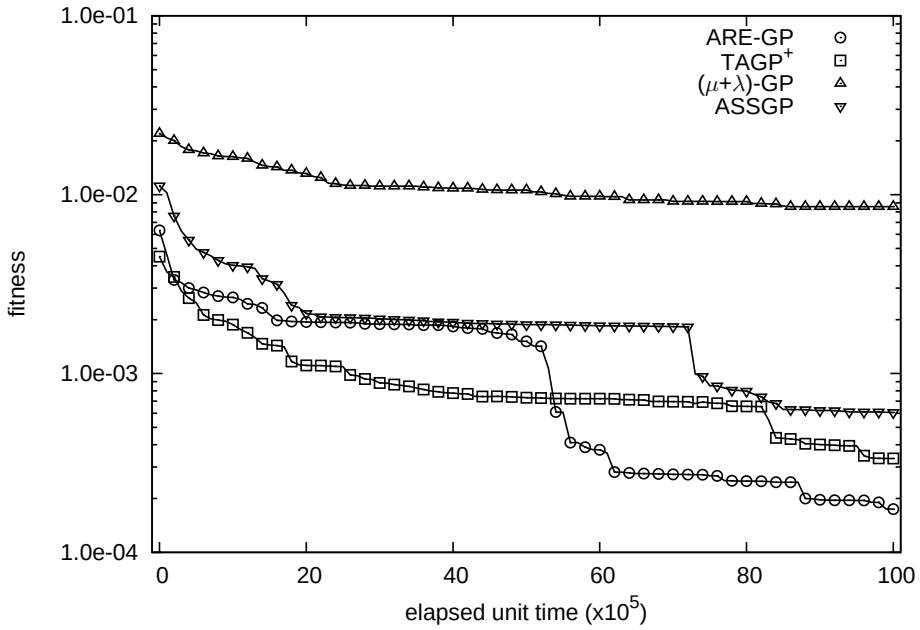


(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 8.6　Case1：同一の評価回数における平均適合度の変化（4/4）

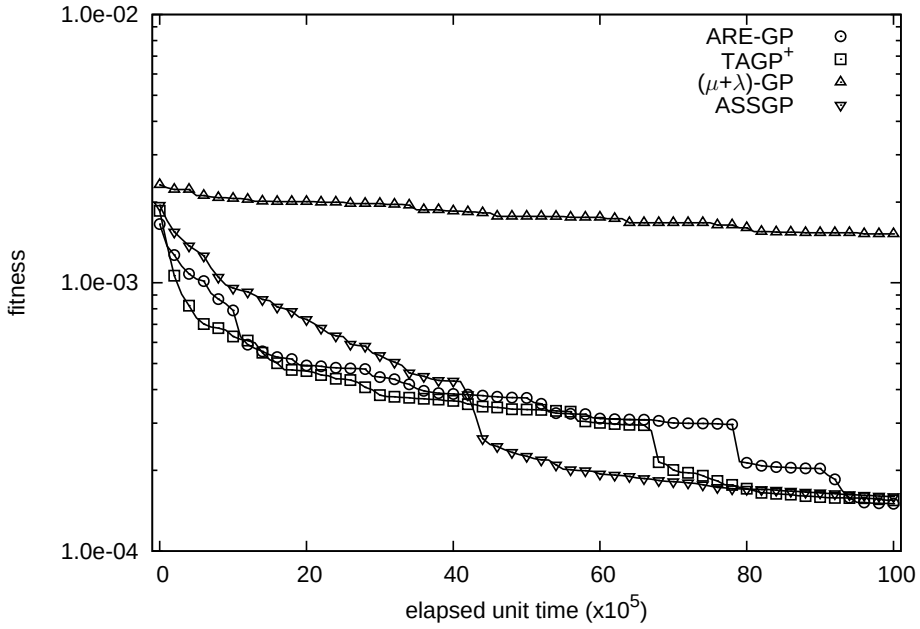


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

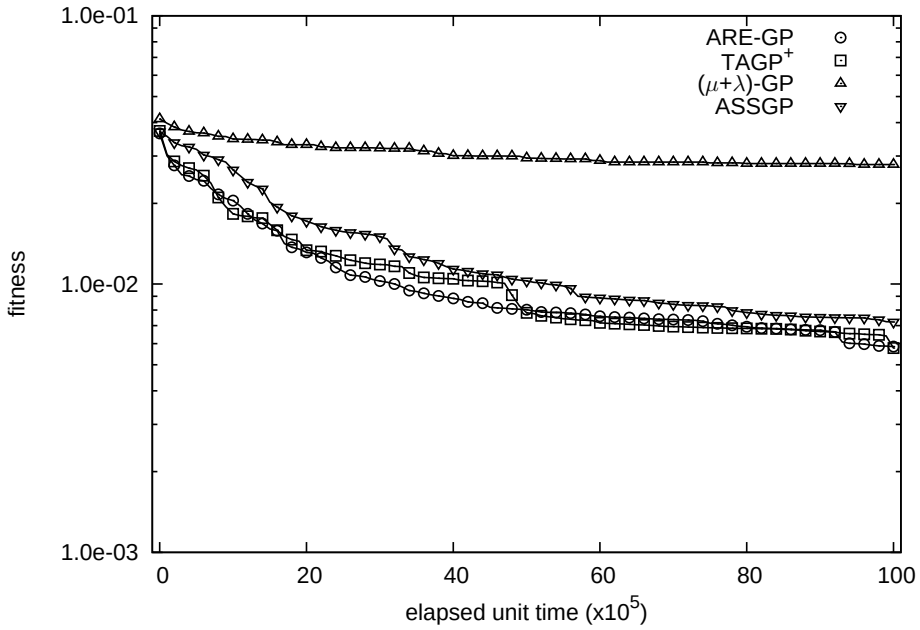


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 8.7　Case1：同一の経過単位時間における平均適合度の変化（1/4）

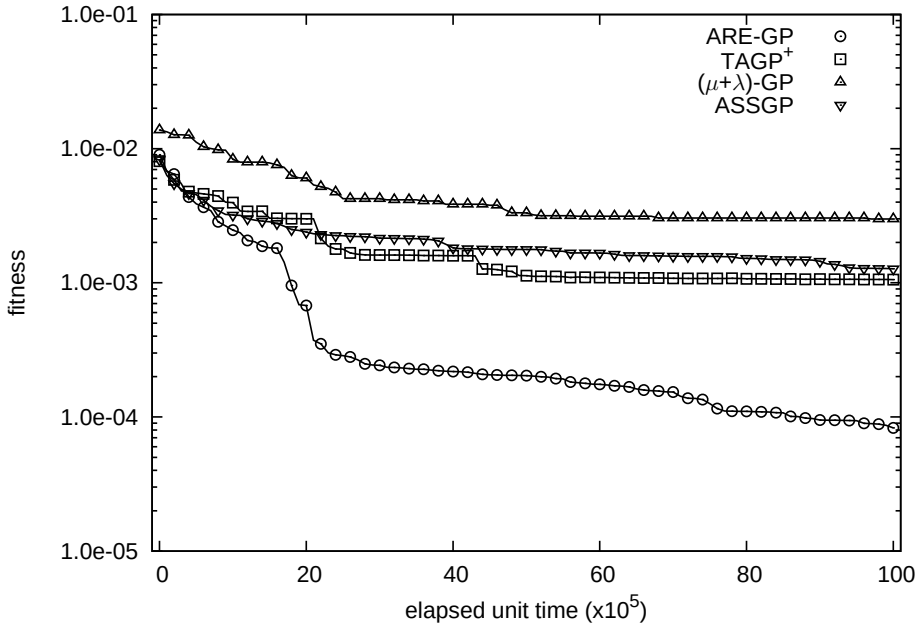


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

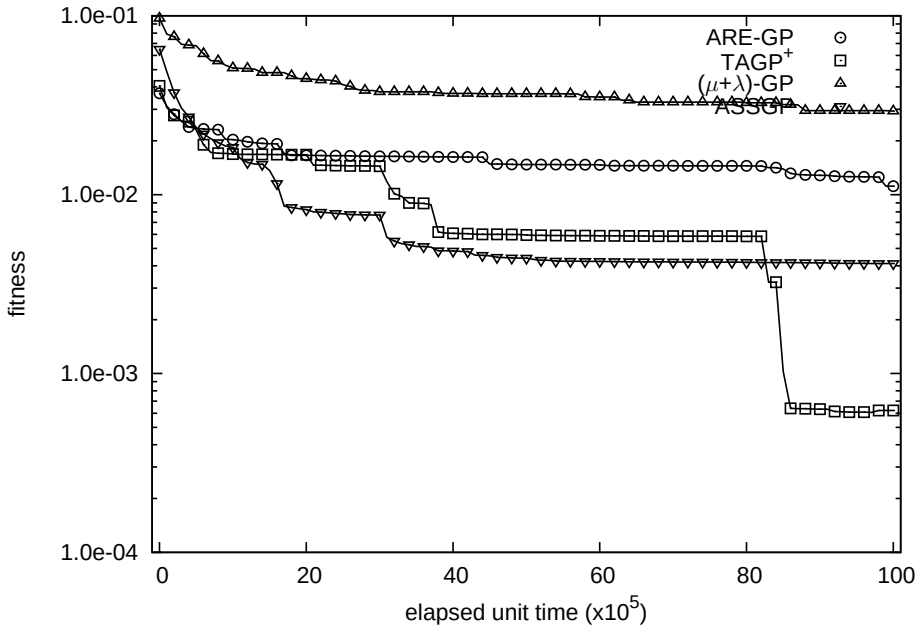


(d) R4（$f(x, y) = x^y$）

図 8.7　Case1：同一の経過単位時間における平均適合度の変化（2/4）

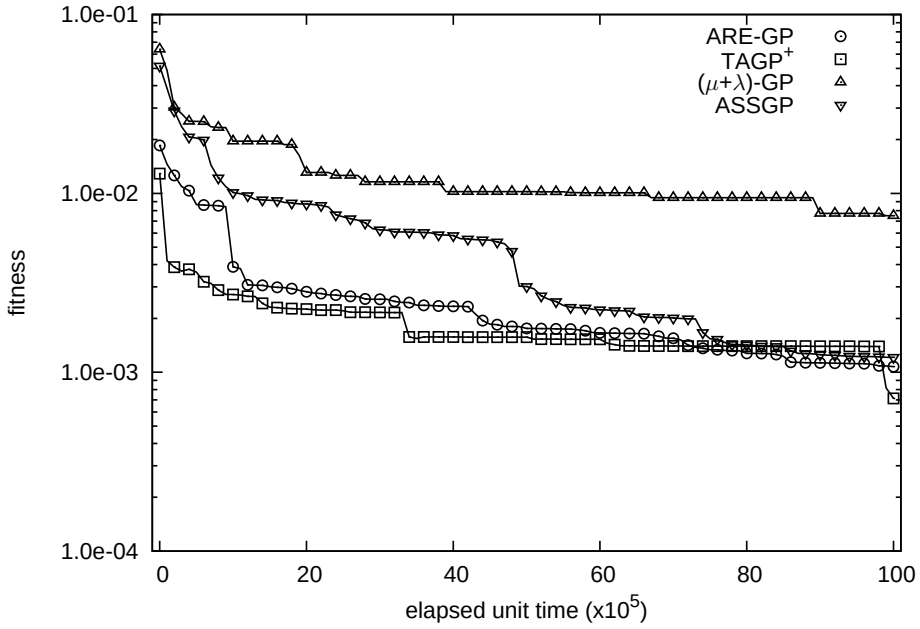


(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

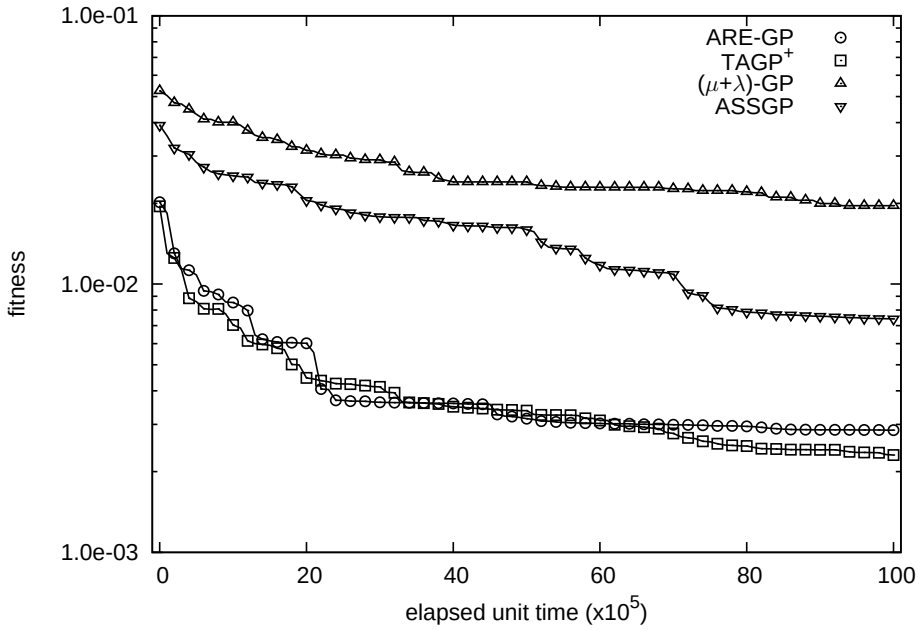


(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 8.7　Case1：同一の経過単位時間における平均適合度の変化（3/4）

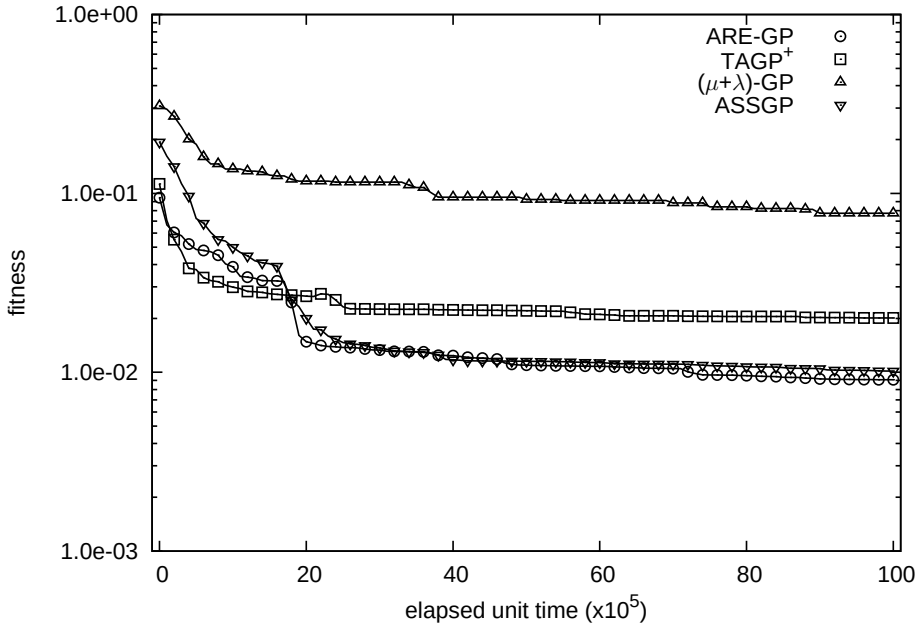


(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）

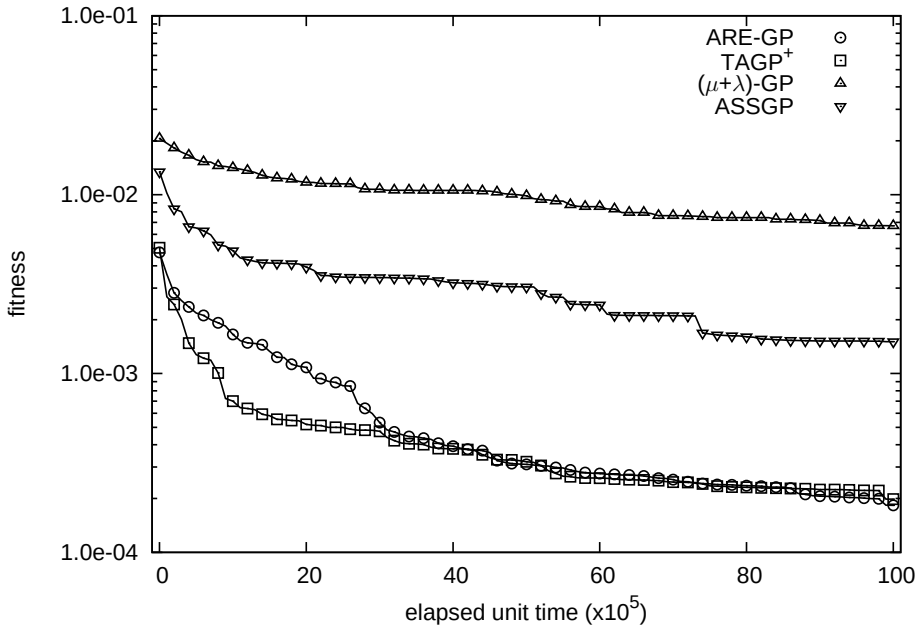


(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 8.7　Case1：同一の経過単位時間における平均適合度の変化（4/4）

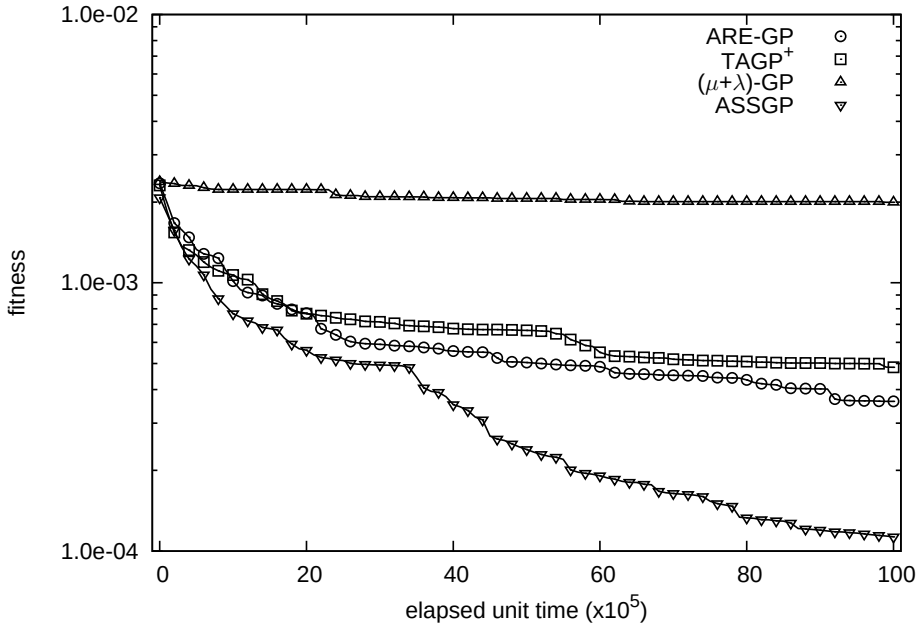


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

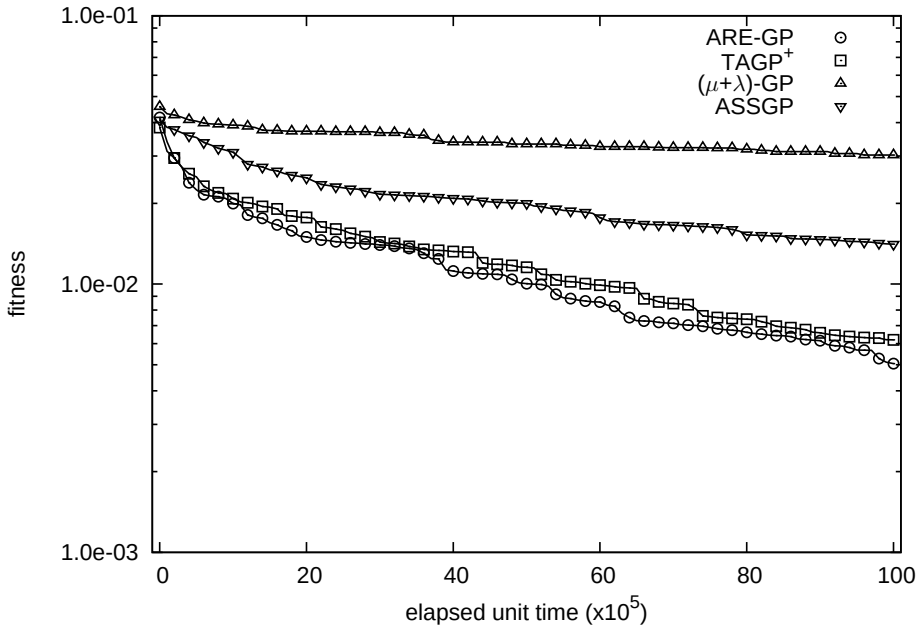


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 8.8　Case2：同一の経過単位時間における平均適合度の変化（1/4）

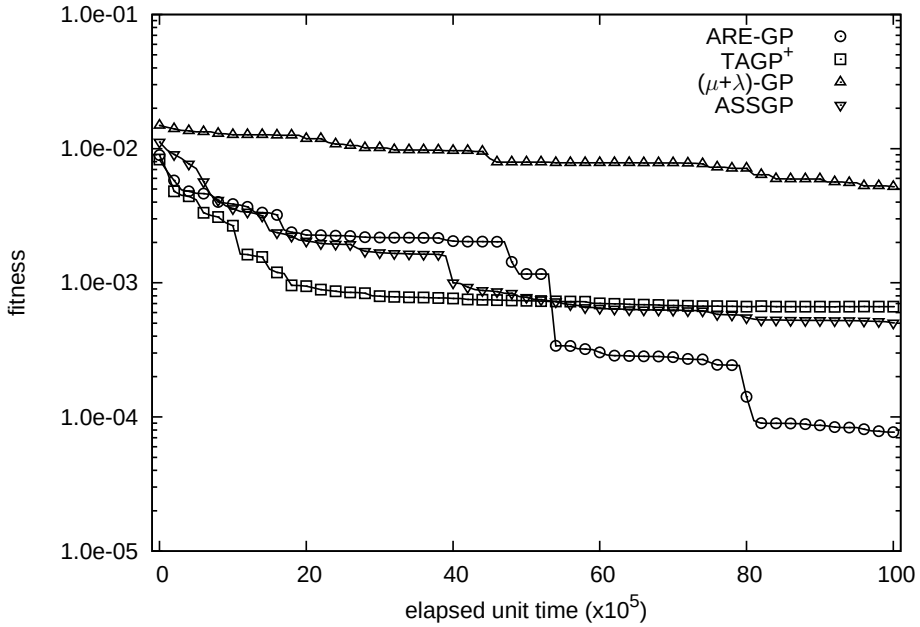


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

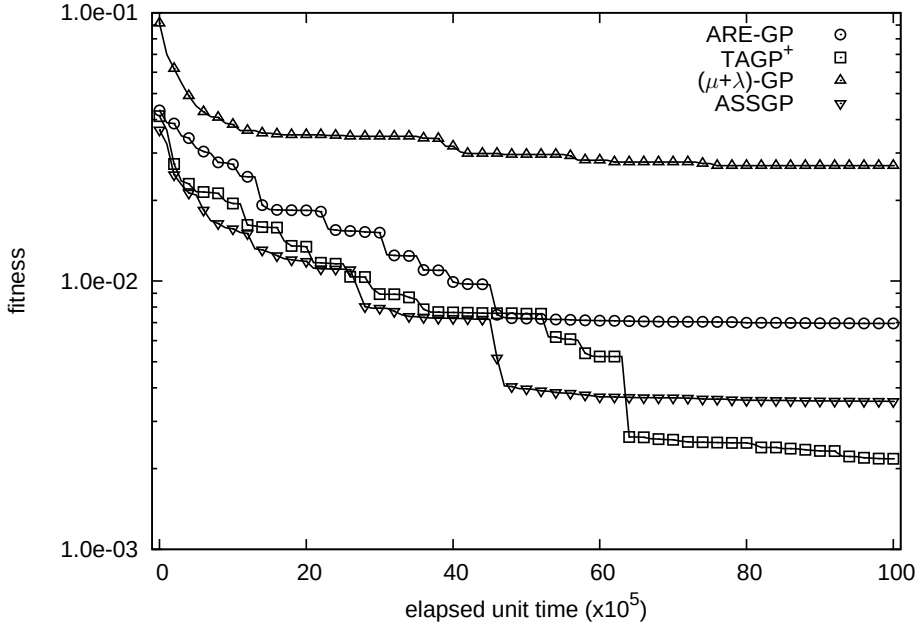


(d) R4（$f(x, y) = x^y$）

図 8.8　Case2：同一の経過単位時間における平均適合度の変化（2/4）

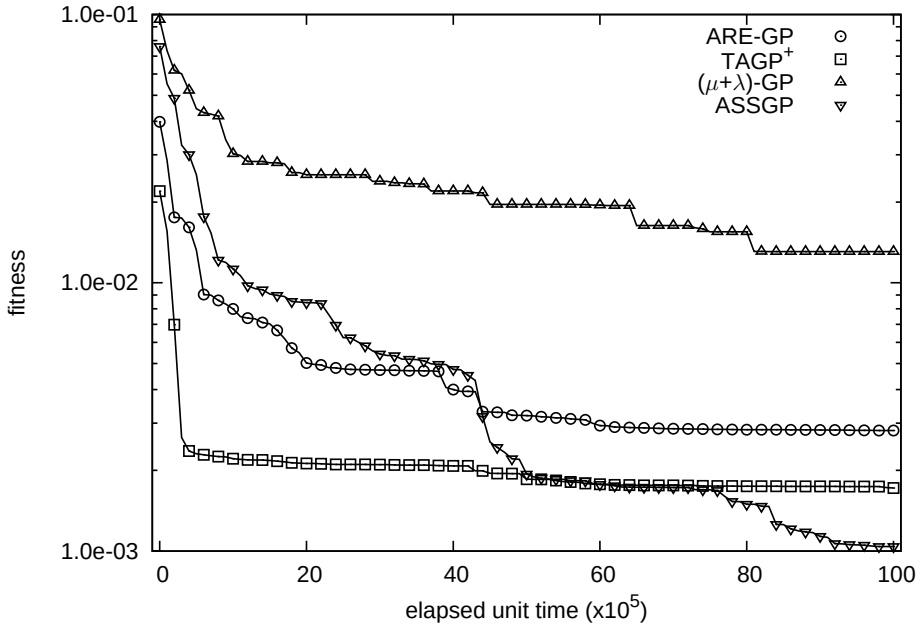


(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

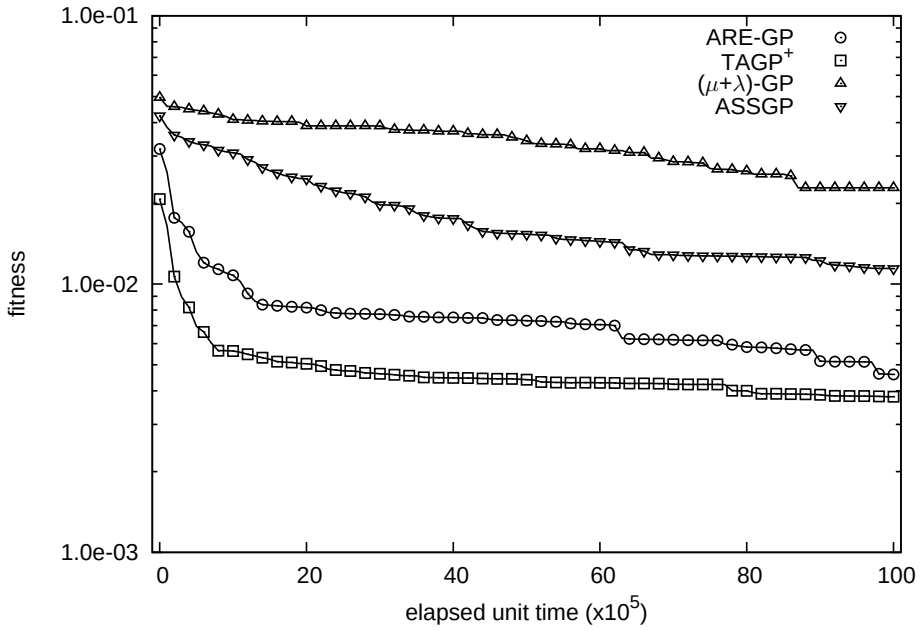


(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 8.8　Case2：同一の経過単位時間における平均適合度の変化（3/4）



(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）



(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 8.8　Case2：同一の経過単位時間における平均適合度の変化（4/4）

せることなく進化が可能であり，従来の非同期 GP である ASSGP と同等以上の性能を示すといえる．

Case3：評価が完了しない個体が含まれる場合

評価が完了しない個体が含まれる Case3 の同一の経過単位時間における平均適合度の変化を図 8.9 に示す．図 8.9 において，各軸，各プロットは図 8.7 と同一である．本実験では，個体（プログラム）中の最大命令数を 256 命令，単位時間あたりの実行可能命令数を 100 命令，データ数を 100 個としているため，最適な待機時間は 256 単位時間となる．各例題において，最大待機時間は 256 単位時間に設定しており，その時間を超えた個体の評価値は $-\infty$ とする．なお比較のために，最適値の 10 倍に当たる 2560 単位時間の結果も示している（図 8.9 中の塗りつぶしのプロット）．

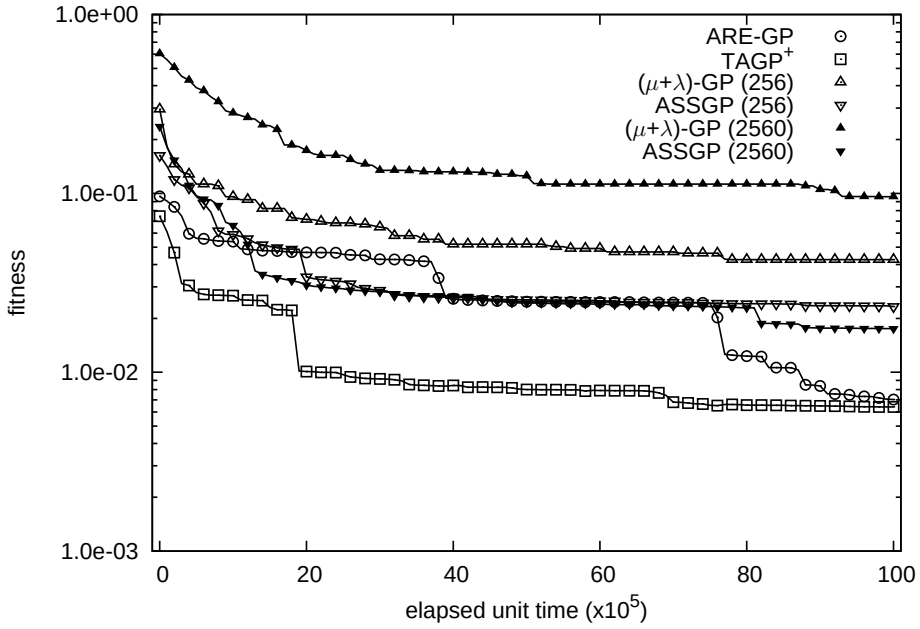
ARE-GP の結果を見ると，上限値を最適に設定した $(\mu + \lambda)$-GP，ASSGP と比べても優れた性能を示しており，かつすべての個体の評価が完了する Case1 と同等の結果が得られている．

この結果から，ARE-GP は評価が完了しない個体が含まれる場合でも同期進化のような待機時間の上限を必要とせずに性能を低下させることなく進化が可能であることが示された．

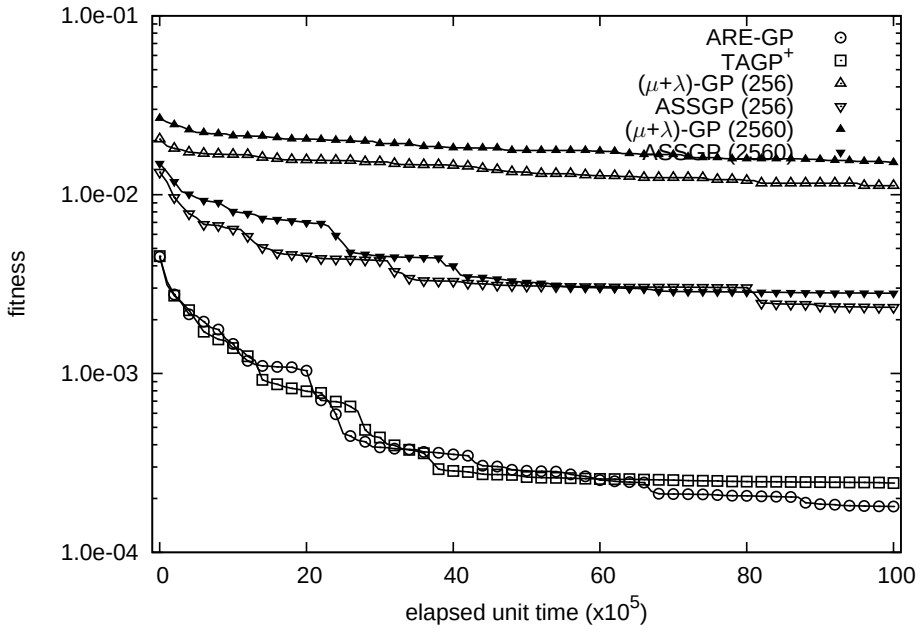
Case4：Case2 と Case3 が同時に起こる場合

Case2 と Case3 が同時に起こる Case4 の同一の経過単位時間における平均適合度の変化を図 8.10 に示す．図 8.10 において，各軸，各プロットは図 8.7 と同一である．本実験では各個体の計算速度にばらつきがあるため，$(\mu + \lambda)$-GP，ASSGP における待機時間は単位時間あたりの実行可能命令数が最も少ない 20 命令/単位時間に上限値を合わせた 1280 単位時間の設定を用いる．また，上限値を超えた個体の評価値は Case3 と同様に $-\infty$ とする．

結果から，ARE-GP は Case4 においても収束速度，最終世代の適合度ともに，Case1 からの性能の低下を抑えれていることがわかる．また，ARE-GP と ASSGP を比較すると，ARE-GP が R6 を除く全ての例題で収束速度，最終世代の適合度とも ASSGP を上回る性能を示していることがわかる．特に，ASSGP は待機時間の上限値を最適値に設定していることから最適値を設定できない場合に Case3 と同様に性能が低下する．このことから，ARE-GP は各個体の計算速度にばらつきがあり，かつ評価が完了しない個体が含まれるような同期進化では並列計算の効率が著しく低下してしまう環境においても性能を低下させることなく進化が可能であり，かつ従来の非同期 GP である ASSGP を上回る性能を示すことが明らかになった．

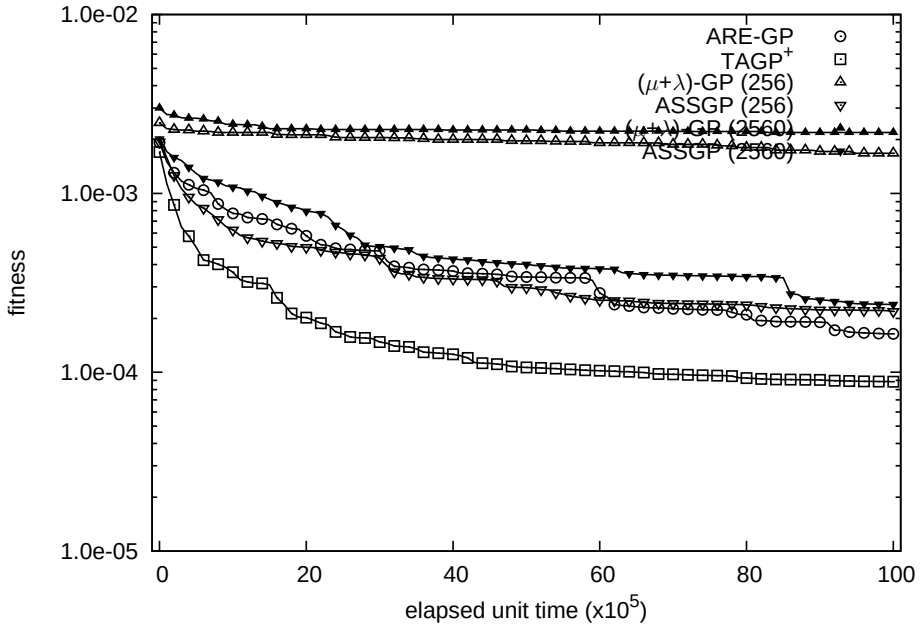


(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）

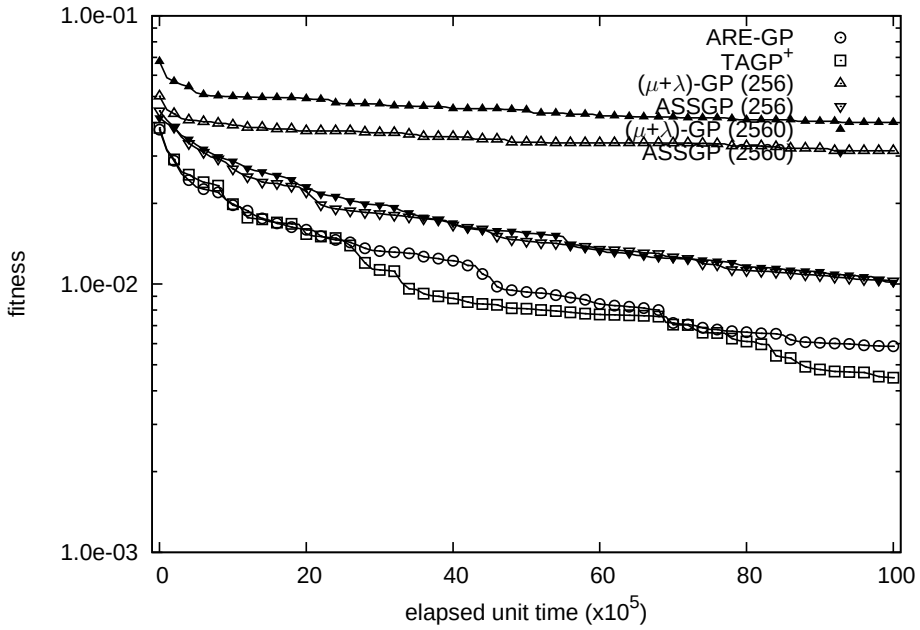


(b) R2（$f(x) = x^5 - 2x^3 + x$）

図 8.9　Case3：同一の経過単位時間における平均適合度の変化（1/4）

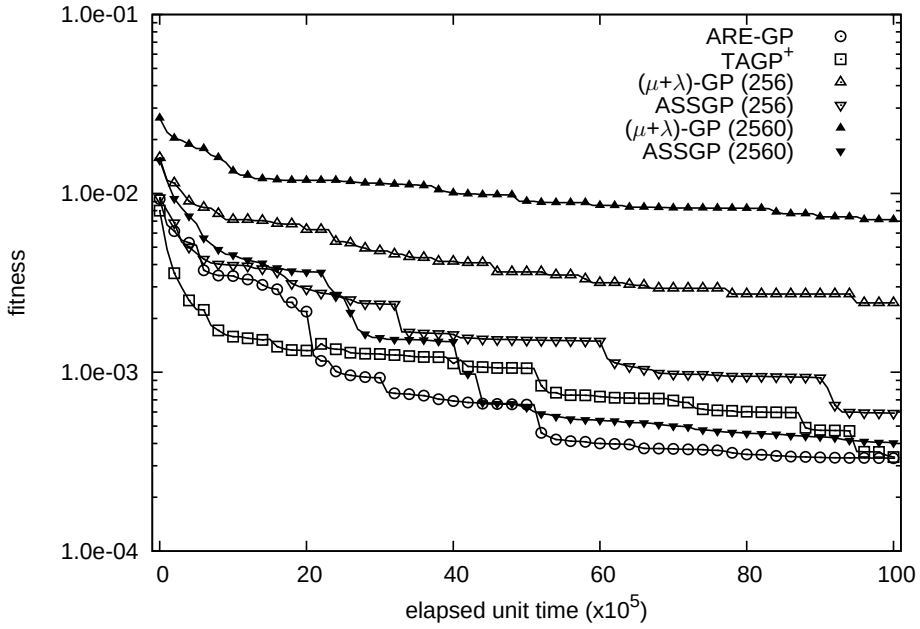


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

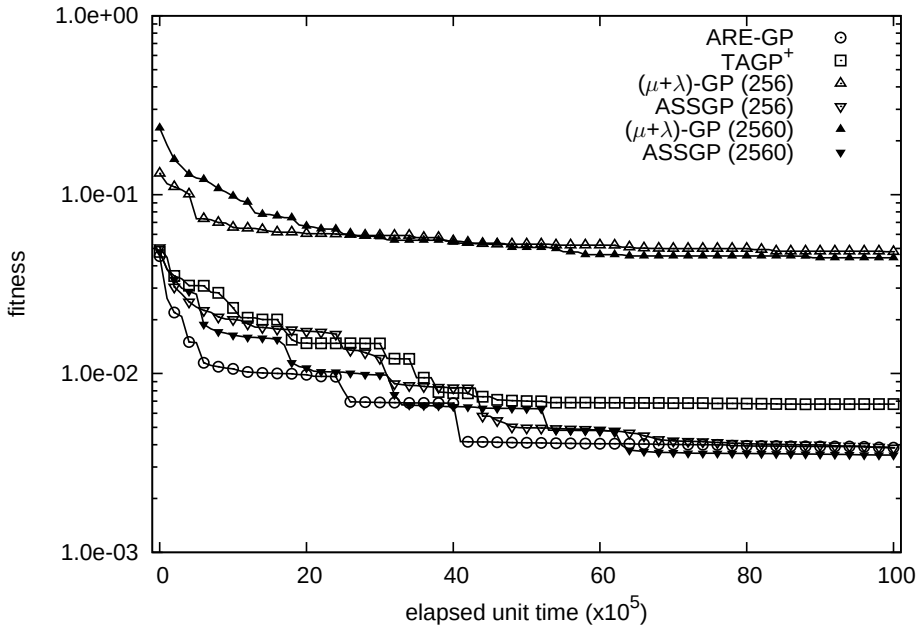


(d) R4（$f(x, y) = x^y$）

図 8.9　Case3：同一の経過単位時間における平均適合度の変化（2/4）

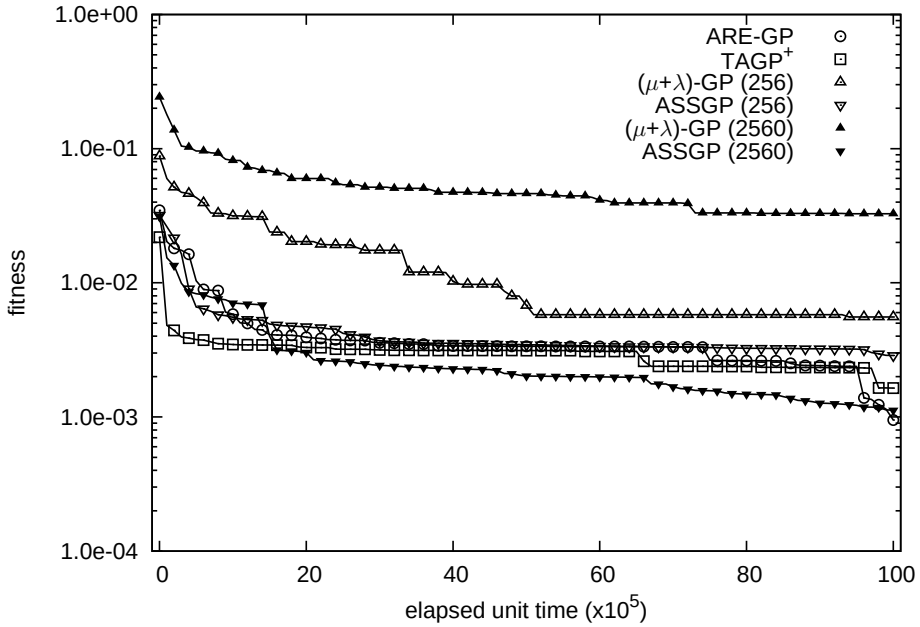


(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）

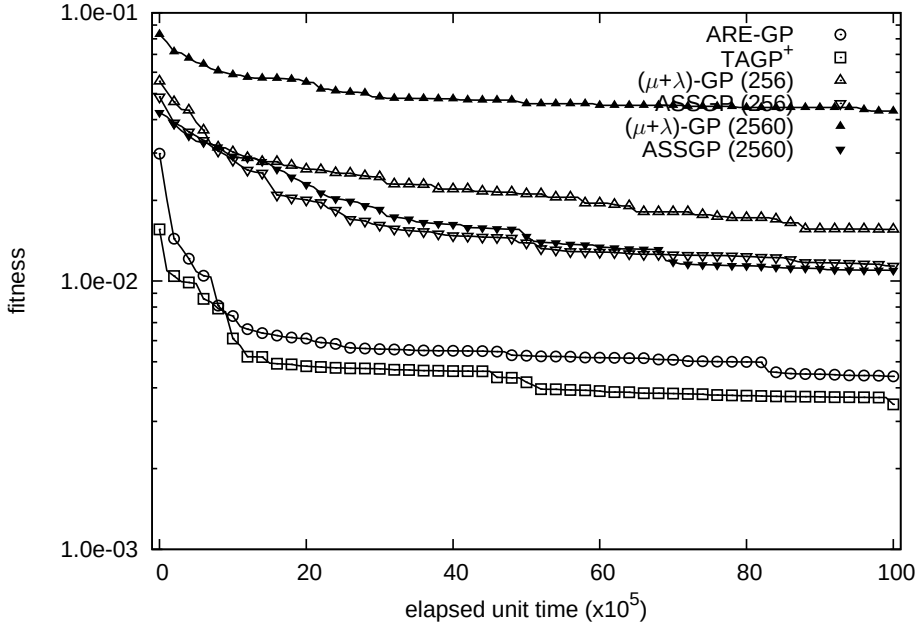


(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

図 8.9　Case3：同一の経過単位時間における平均適合度の変化（3/4）

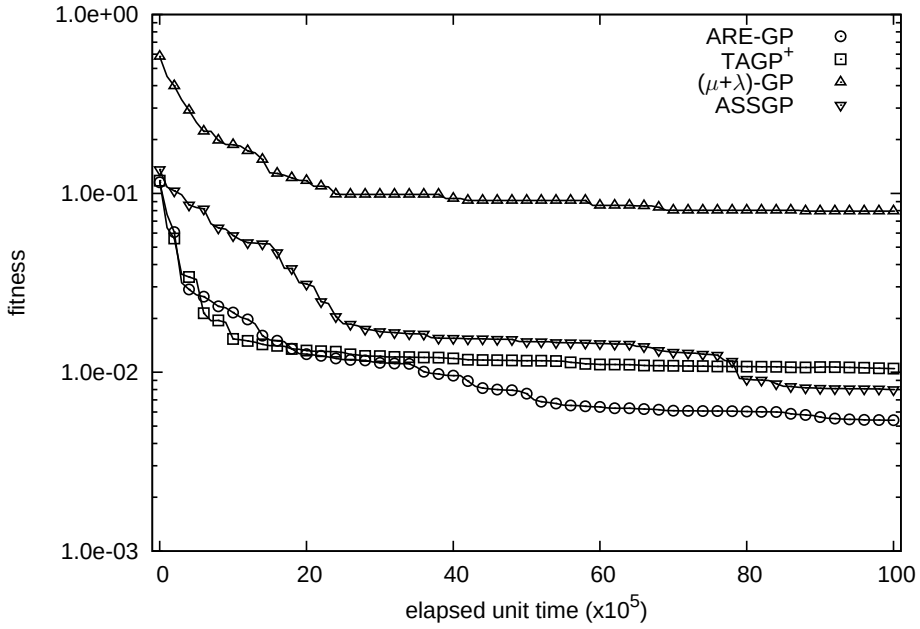


(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）

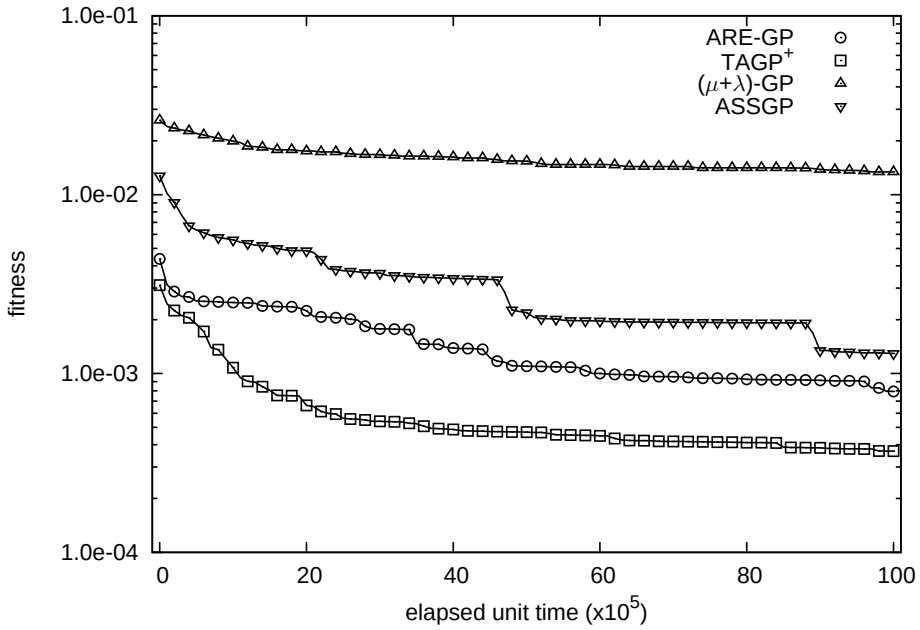


(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 8.9　Case3：同一の経過単位時間における平均適合度の変化（4/4）

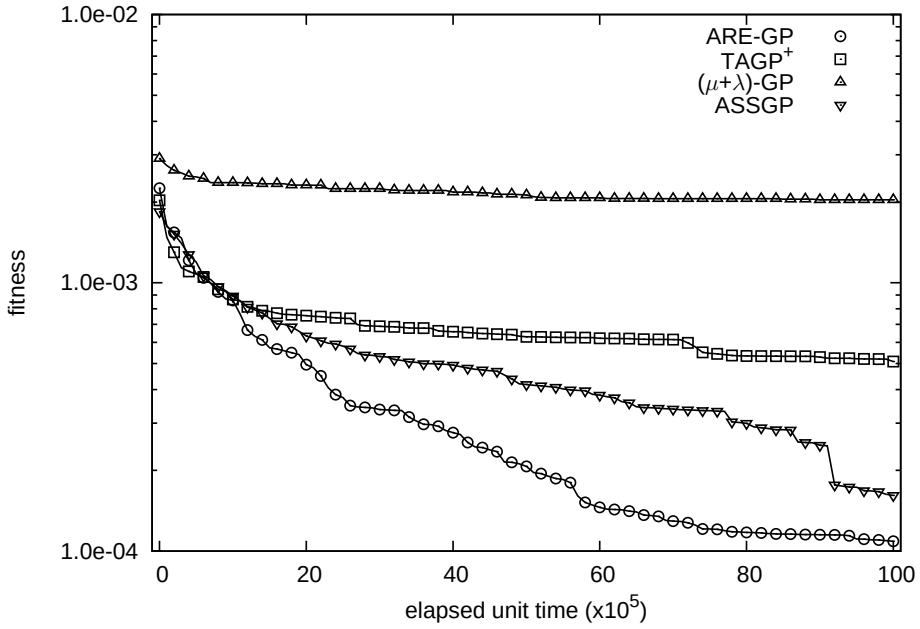


(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）

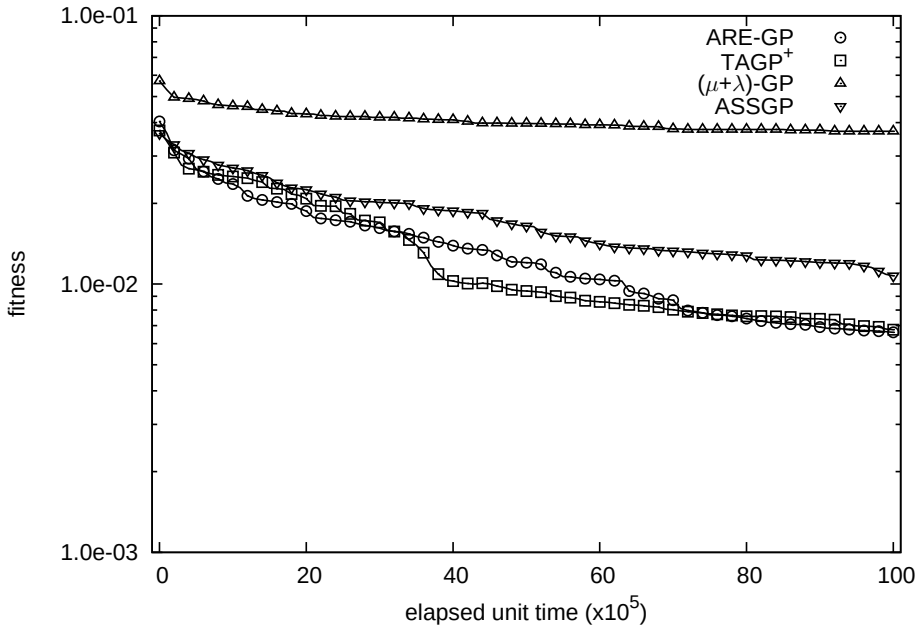


(b) R2（$f(x) = x^5 - 2x^3 + x$）

図 8.10　Case4：同一の経過単位時間における平均適合度の変化（1/4）

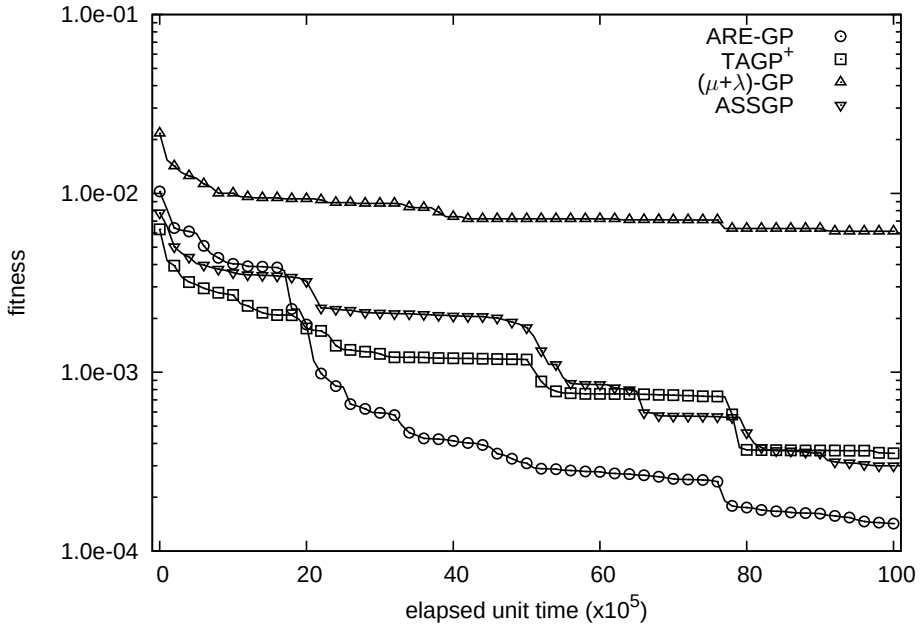


(c) R3 ($f(x) = x^6 - 2x^4 + x^2$)

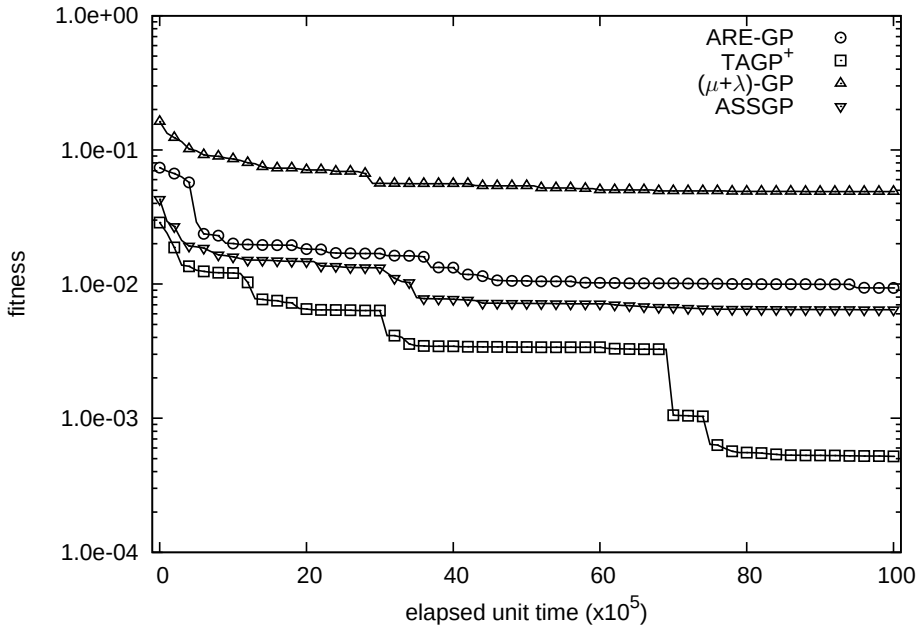


(d) R4 ($f(x, y) = x^y$)

図 8.10　Case4：同一の経過単位時間における平均適合度の変化（2/4）

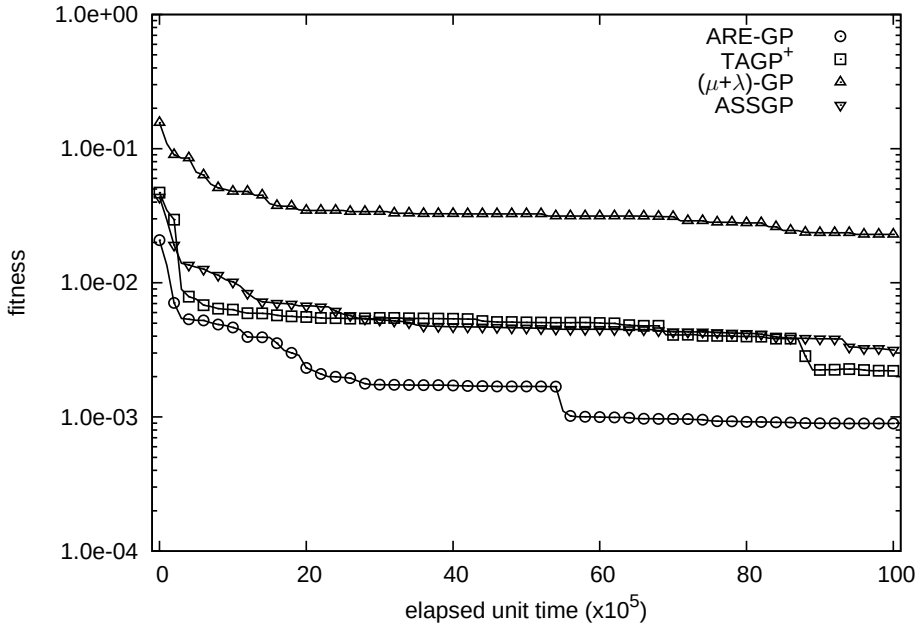


(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）

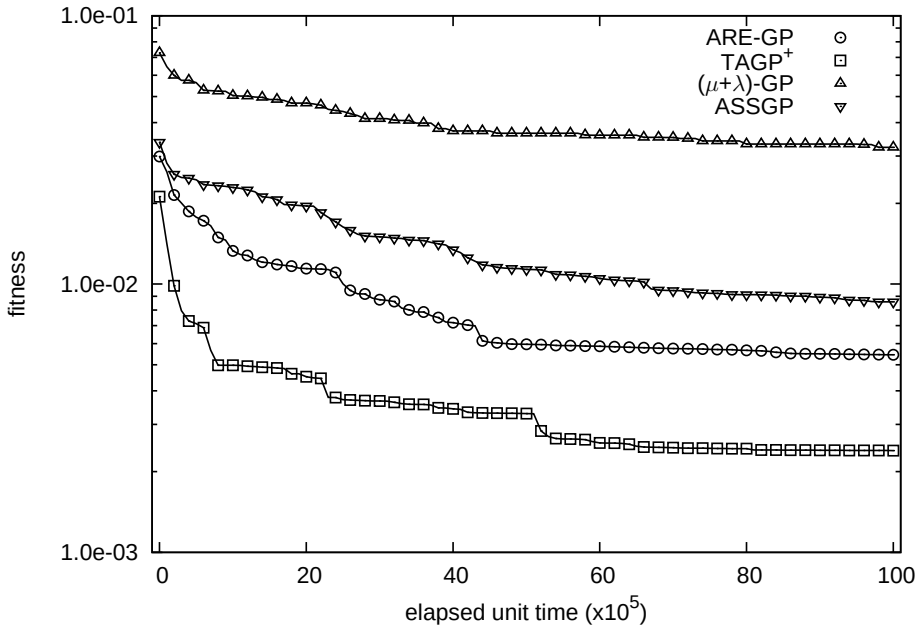


(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

図 8.10　Case4：同一の経過単位時間における平均適合度の変化（3/4）



(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）



(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

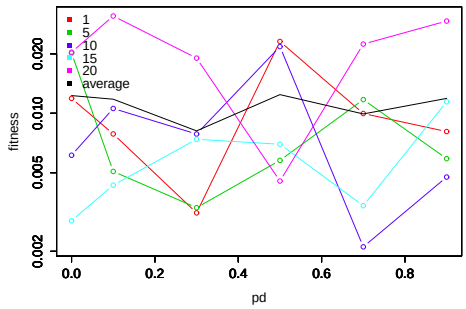
図 8.10　Case4：同一の経過単位時間における平均適合度の変化（4/4）

8.4.4 考察 1：パラメータ分析

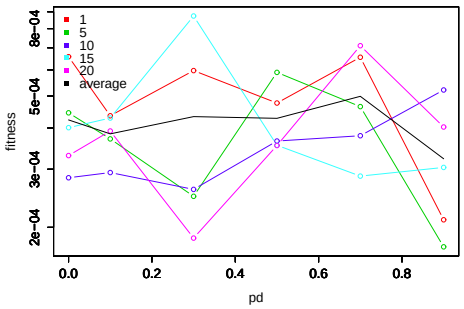
ARE-GP において，適合度削除確率 P_d とアーカイブサイズ as の影響を分析するために，それぞれを $P_d = \{0.0, 0.1, 0.3, 0.5, 0.7, 0.9\}$，$as = \{1, 5, 10, 15, 20\}$ に設定し，Case1，および Case4 において各例題，各パラメータの組み合わせを 30 試行ずつ行う．なお，$P_d = 0.0$ の結果は適合度削除は実行されずにリーパー削除のみによって個体削除が行われることを意味する．$P_d = 0.9$，$as = 5$ の結果は前節での結果と同一である．

はじめに，適合度削除確率 P_d の影響を分析するために，図 8.11，8.13，8.12，8.14 に各適合度削除確率 P_d においてアーカイブサイズ $as = \{1, 5, 10, 15, 20\}$ に設定した際の適合度の平均とそれらのプログラムサイズの平均を示す．図 8.11，8.12 において横軸は適合度削除確率 P_d を表し，縦軸は適合度を表す．図 8.13，8.14 において横軸は適合度削除確率 P_d を表し，縦軸はプログラムサイズを表す．色の違いはアーカイブサイズの違いを表し，黒線は全アーカイブサイズの結果の平均を表す．

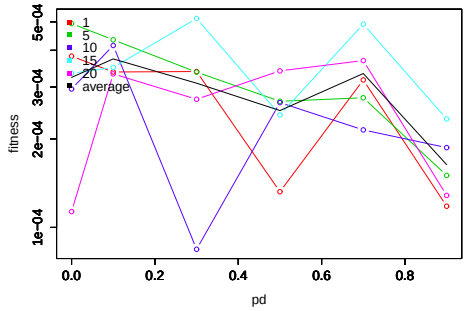
まず適合度について，図 8.11，8.12 の結果から，明確な傾向は見られないものの，特に Case4 においては R8 を除くすべての例題で P_d が高い（$P_d \geq 0.5$）時に適合度の低い（優れた）個体が多く生成されていることがわかる．このことから，特に評価時間差の大きい場合に適合度削除確率は大きく設定することが有効であるといえる．次にプログラムサイズについて，図 8.13，8.14 の結果から，すべての例題において P_d が小さいほどプログラムサイズの小さな個体が生成される傾向があることがわかる．特に，評価時間のばらつきが大きい Case4 の方がより P_d の違いによるプログラムサイズの差が大きくなることがわかる．本実験においては各個体の評価時間はプログラムサイズに依存するため，プログラムサイズが大きな個体が生成されるほど評価時間の長い個体が生成されている事を示している．このことから，P_d の違いによるプログラムサイズの傾向は，P_d が大きいほどより評価時間の長い個体が生成されていることを示している．これは，P_d が大きくなることによって適合度削除の割合が増加するため，もう一つの削除法であるリーパー削除の割合が減少し，評価時間の長い（プログラムサイズの大きい）個体が削除されずに評価を完了できるためである．これらのことから，評価時間の長い個体を待機する必要がある場合には適合度削除確率を高い値に設定する必要があることが明らかになった．適合度削除確率と適合度の間に傾向が見られない原因として GP 特有の特性が挙げられる．具体的には，GP においては同定誤差（適合度）が 0 となるようなプログラムは比較的小さなプログラムで実現される一方，適合度の改善は小さいままプログラムのサイズが徐々に増加するブロート（*bloat*）と呼ばれる現象によってサイズが非常に大きく（本実験では最大プログラムサイズの 256 に近い）適合度の高い個体が生成される．これによって，P_d が小さい場合にはプログラムサイズが小さく適合度の高い個体が生成され，逆に P_d が大きい場合にはブロートによってサイズが大きく適合度の高い個体が生成されるため適合度に対



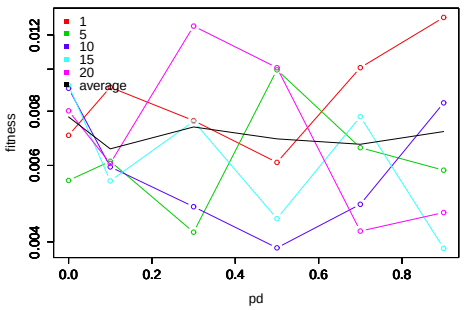
(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)



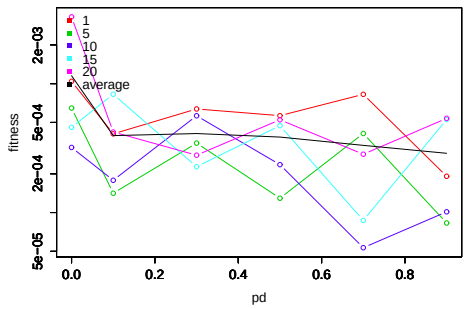
(b) R2 ($f(x) = x^5 - 2x^3 + x$)



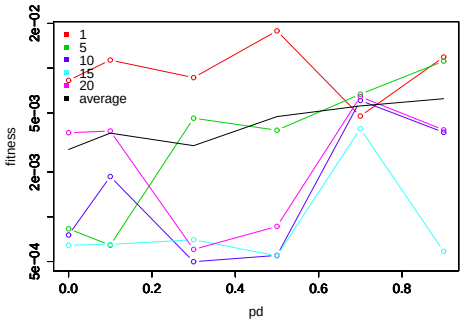
(c) R3 ($f(x) = x^6 - 2x^4 + x^2$)



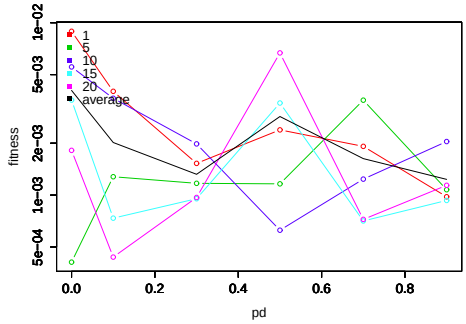
(d) R4 ($f(x, y) = x^y$)



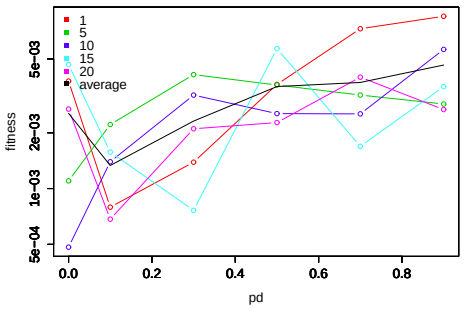
(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)



(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

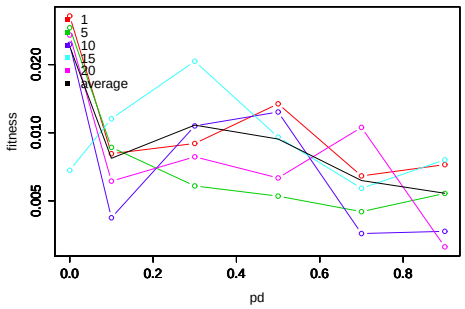


(g) R7 ($f(x) = ln(x + 1) + ln(x^2 + 1)$)

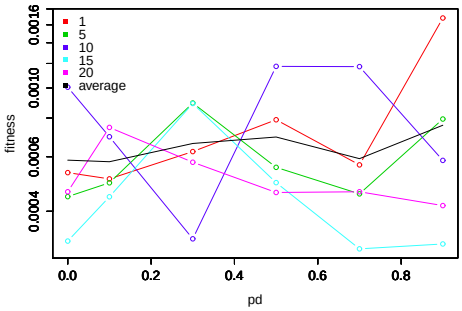


(h) R8 ($f(x, y) = sin(x) + sin(y^2)$)

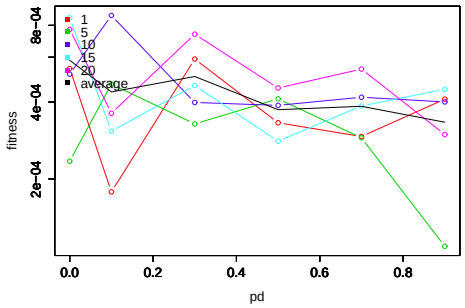
図 8.11　適合度削除確率 P_d の変化による適合度の変化（Case1）



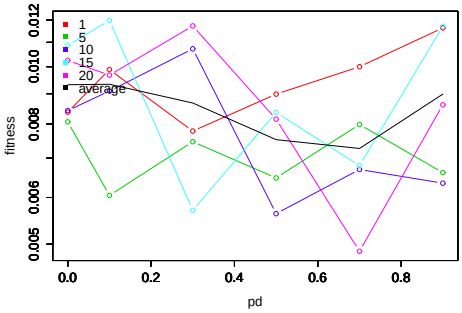
(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)



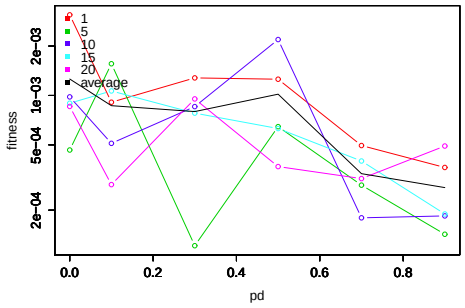
(b) R2 ($f(x) = x^5 - 2x^3 + x$)



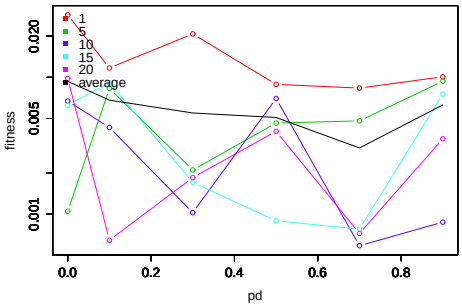
(c) R3 ($f(x) = x^6 - 2x^4 + x^2$)



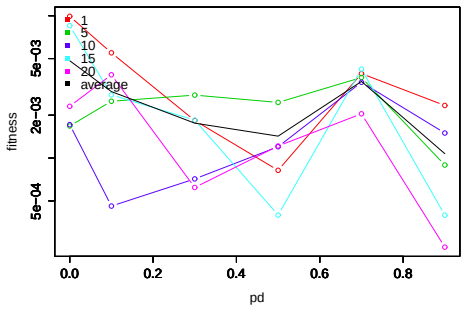
(d) R4 ($f(x, y) = x^y$)



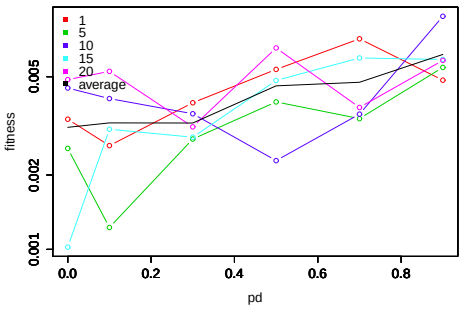
(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)



(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)

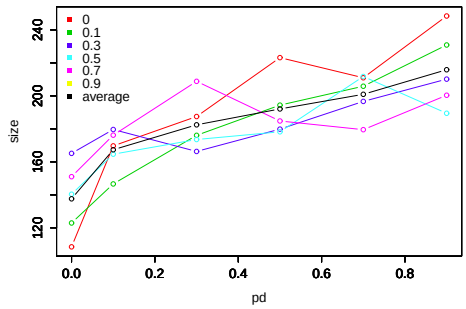


(g) R7 ($f(x) = ln(x + 1) + ln(x^2 + 1)$)

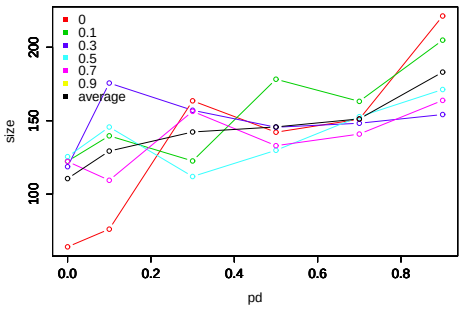


(h) R8 ($f(x, y) = sin(x) + sin(y^2)$)

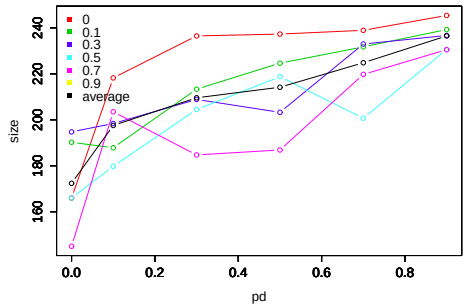
図 8.12　適合度削除確率 P_d の変化による適合度の変化（Case4）



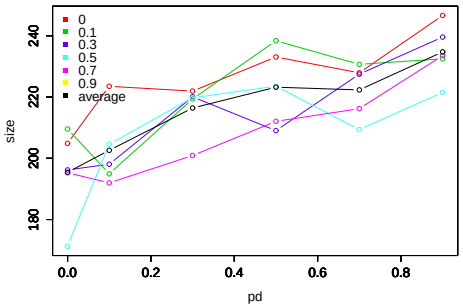
(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）



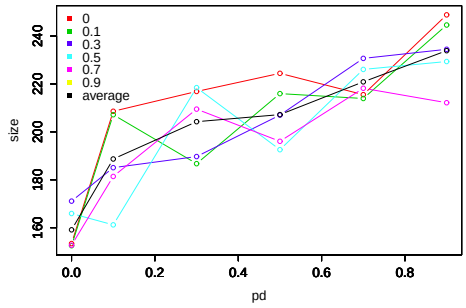
(b) R2（$f(x) = x^5 - 2x^3 + x$）



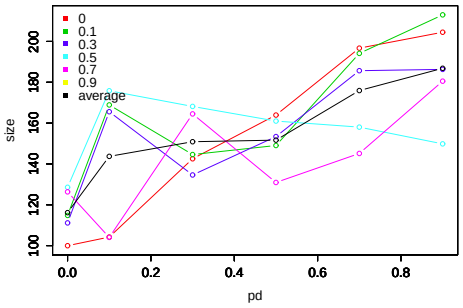
(c) R3（$f(x) = x^6 - 2x^4 + x^2$）



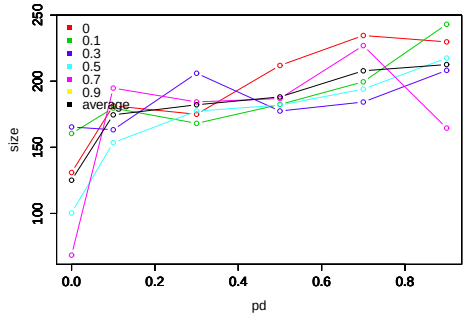
(d) R4（$f(x, y) = x^y$）



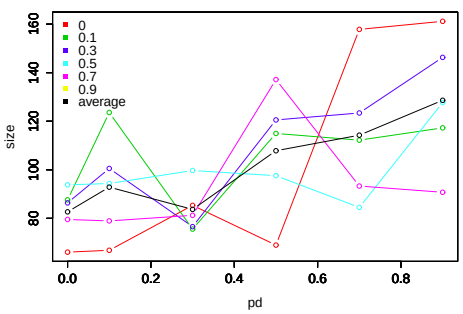
(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）



(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

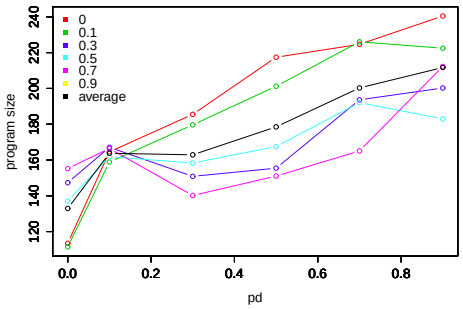


(g) R7（$f(x) = ln(x + 1) + ln(x^2 + 1)$）

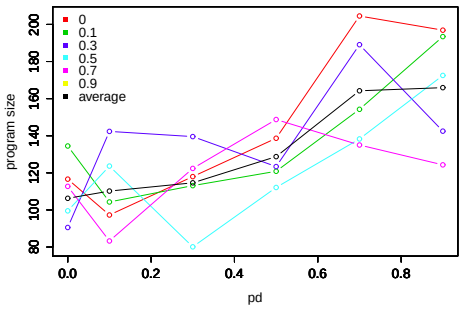


(h) R8（$f(x, y) = sin(x) + sin(y^2)$）

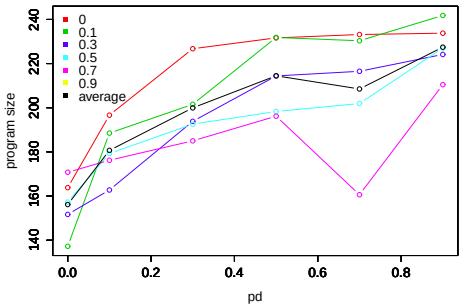
図 8.13　適合度削除確率 P_d の変化によるプログラムサイズの変化（Case1）



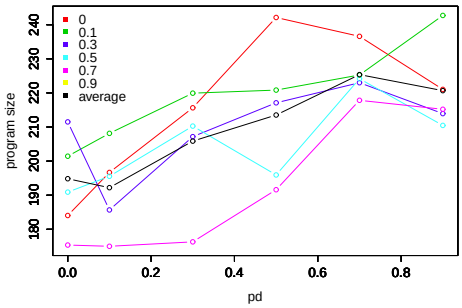
(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)



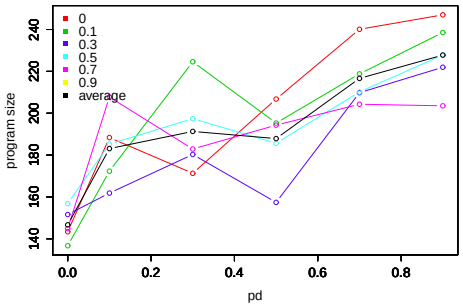
(b) R2 ($f(x) = x^5 - 2x^3 + x$)



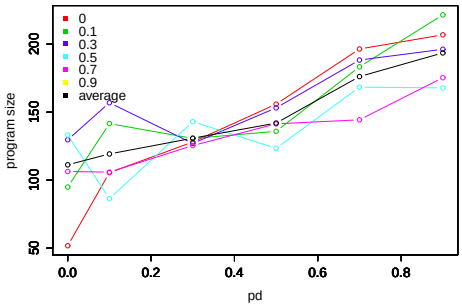
(c) R3 ($f(x) = x^6 - 2x^4 + x^2$)



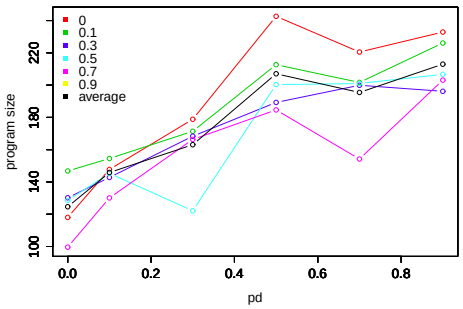
(d) R4 ($f(x, y) = x^y$)



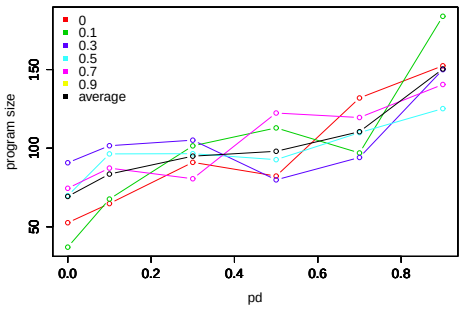
(e) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)



(f) R6 ($f(x) = sin(x) + sin(x + x^2)$)



(g) R7 ($f(x) = ln(x + 1) + ln(x^2 + 1)$)



(h) R8 ($f(x, y) = sin(x) + sin(y^2)$)

図 8.14　適合度削除確率 P_d の変化によるプログラムサイズの変化（Case4）

する傾向が見られないと考えられる．

次に，アーカイブサイズ as の影響を分析するために，図 8.15，8.17，8.16，8.18 に各適合度削除確率 P_d においてアーカイブサイズ $as = \{1, 5, 10, 15, 20\}$ に設定した際の適合度の平均とそれらのプログラムサイズの平均を示す．図 8.15，8.16 において横軸はアーカイブサイズ as を表し，縦軸は適合度を表す．図 8.17，8.18 において横軸はアーカイブサイズ as を表し，縦軸はプログラムサイズを表す．色の違いは適合度削除確率の違いを表し，黒線は全適合度削除確率の結果の平均を表す．

まず適合度について，図 8.15，8.16 の結果から，すべての例題においてアーカイブサイズについての有意な傾向は見られなかった．このことから，ARE-GP ではアーカイブサイズの設定は性能に大きく影響を与えないことが明らかになった．これは，特に GP のように単一目的最適化の場合にはアーカイブサイズによらずアーカイブが最適個体でうめつくされるためである．次にプログラムサイズについて，図 8.17，8.18 の結果から，いくつかの例題においてアーカイブサイズが大きいほどプログラムサイズの小さな個体が生成される傾向があることがわかる．これらのことから，ARE-GP においては比較的小さなアーカイブサイズで十分な性能が得られるといえる．

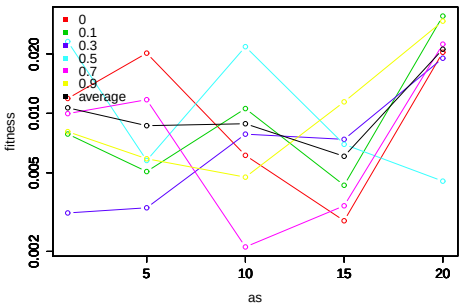
以上の結果をまとめると，ARE-GP においては適合度削除確率 P_d は特に評価時間の長い個体を待機する必要がある場合には 0.5 以上の大きな値に設定することが有効であり，アーカイブサイズは比較的小さな値に設定する場合でも十分な性能が得られるといえる．

8.4.5 考察 2：評価時間のばらつきによる影響

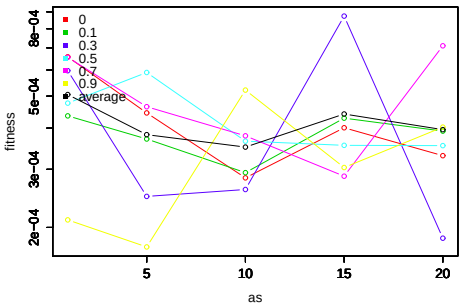
本実験では，評価時間が異なる分散環境における ARE-GP と $(\mu + \lambda)$-GP，ASSGP の性能を比較した．両手法のそれぞれの環境における特徴をまとめる．

$(\mu + \lambda)$-GP は，各個体の計算速度にばらつきがある場合には，最も計算時間の遅い計算機の評価を待機する必要があるため，その分の時間が無駄になってしまい，単位時間あたりの進化性能が低下する．同期進化の場合は，本実験のように計算速度の遅い計算機が極小数であってもその計算機が評価し終わるのを待機する必要があり，その他の計算機がどれだけ短時間で評価が完了しても無駄になってしまう．評価が完了しない個体が含まれる場合には，待機時間の上限値を適切に設定することができれば待機時間を無駄にせずに進化が可能である．しかし，待機時間の上限値が適切に設定できない場合には，無駄な待機時間が生じ，進化性能が低下する．本実験では，待機時間の適切な値が既知であったため設定可能であったが，一般には待機時間の適切な値が未知であることがほとんどであるため，評価が完了しない個体を含む場合には同期進化では待機時間分の無駄を生じる．

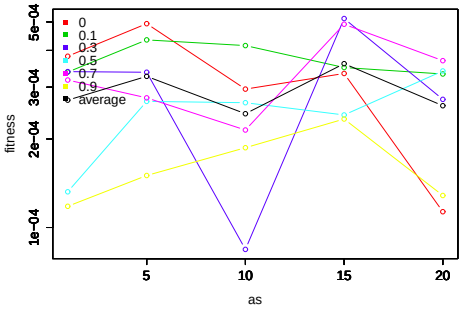
ASSGP は，各個体の計算速度にばらつきがある場合でも，親選択に必要な 4 個体の評価が完了した段階で子個体生成が可能であるため，$(\mu + \lambda)$-GP とくらべて単位時間あたりの解探索性能が優れている．しかし，評価が完了しない個体が含まれる場合には，



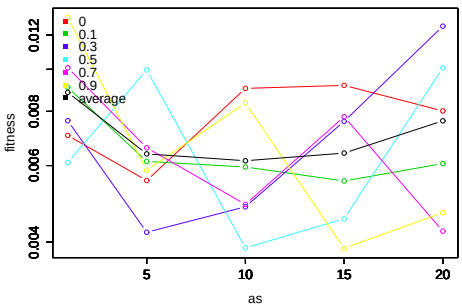
(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）



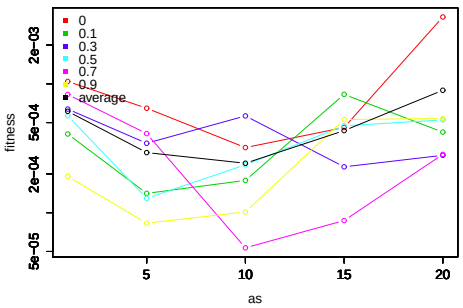
(b) R2（$f(x) = x^5 - 2x^3 + x$）



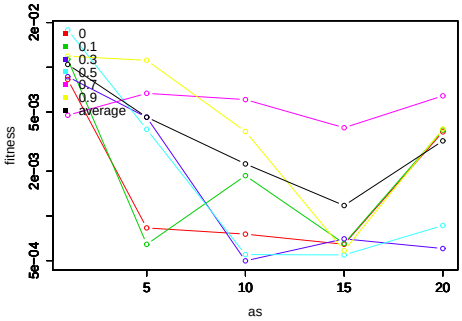
(c) R3（$f(x) = x^6 - 2x^4 + x^2$）



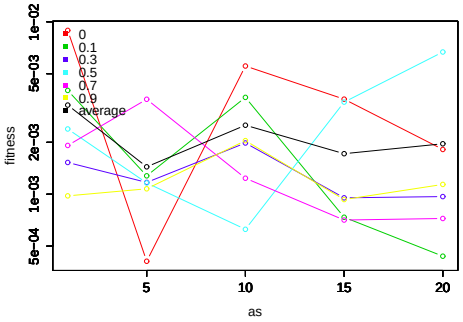
(d) R4（$f(x, y) = x^y$）



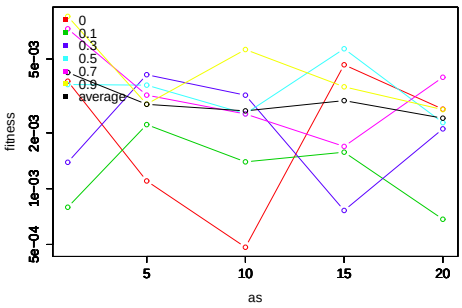
(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）



(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

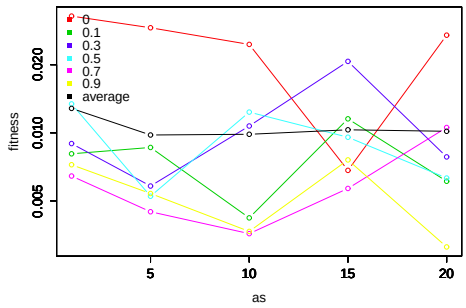


(g) R7（$f(x) = ln(x + 1) + ln(x^2 + 1)$）

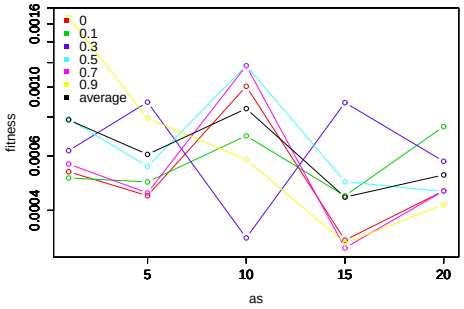


(h) R8（$f(x, y) = sin(x) + sin(y^2)$）

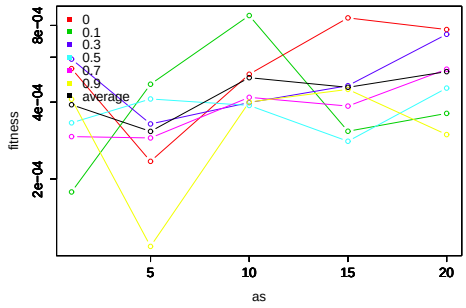
図 8.15　アーカイブサイズ as の変化による適合度の変化（Case1）



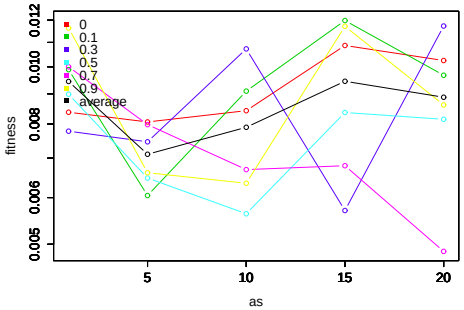
(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）



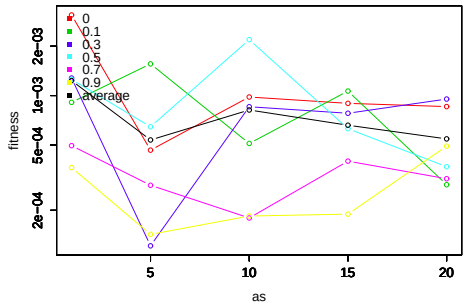
(b) R2（$f(x) = x^5 - 2x^3 + x$）



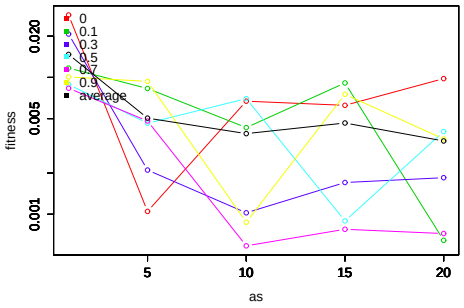
(c) R3（$f(x) = x^6 - 2x^4 + x^2$）



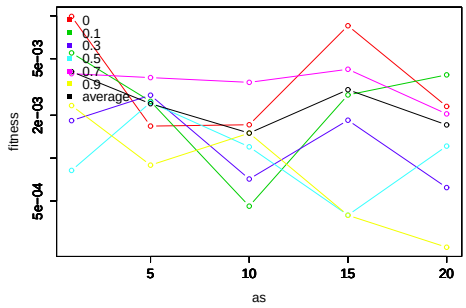
(d) R4（$f(x, y) = x^y$）



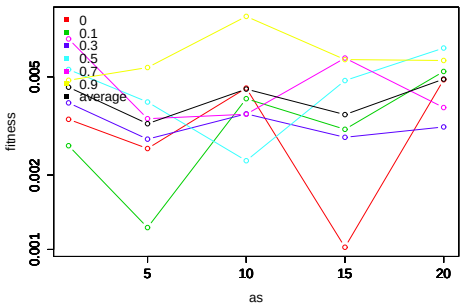
(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）



(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

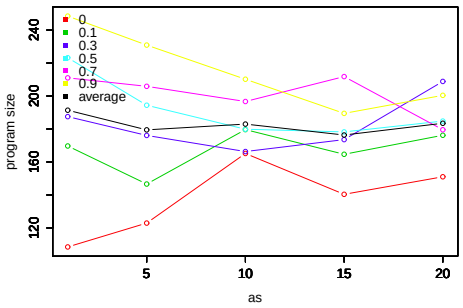


(g) R7（$f(x) = ln(x + 1) + ln(x^2 + 1)$）

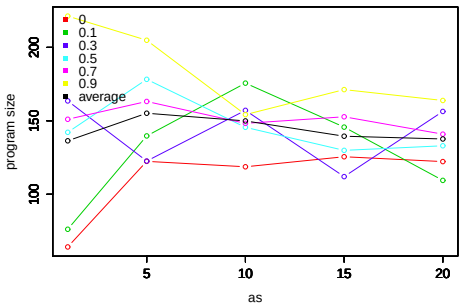


(h) R8（$f(x, y) = sin(x) + sin(y^2)$）

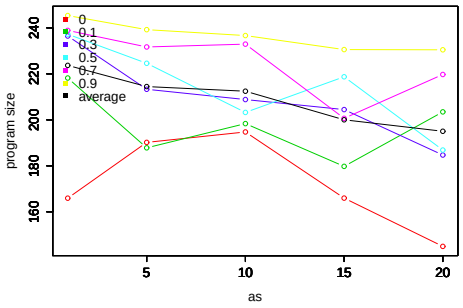
図 8.16　アーカイブサイズ as の変化による適合度の変化（Case4）



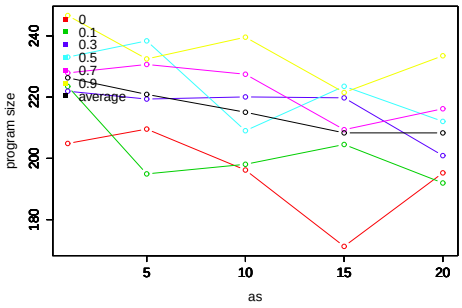
(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）



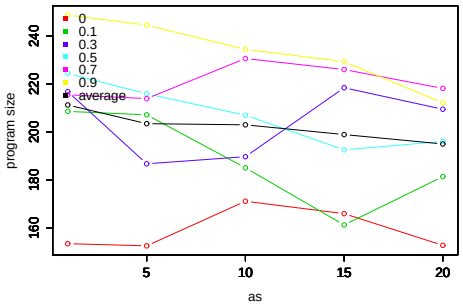
(b) R2（$f(x) = x^5 - 2x^3 + x$）



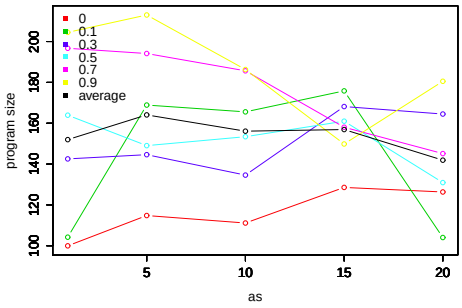
(c) R3（$f(x) = x^6 - 2x^4 + x^2$）



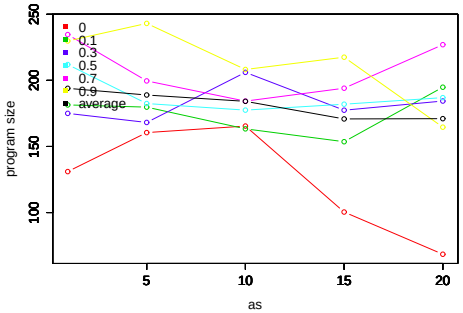
(d) R4（$f(x, y) = x^y$）



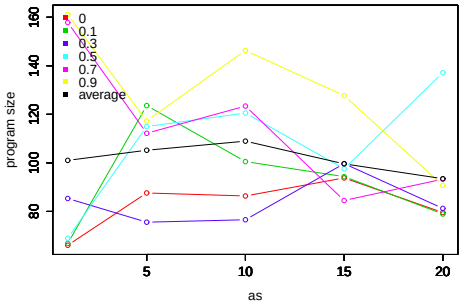
(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）



(f) R6（$f(x) = sin(x) + sin(x + x^2)$）

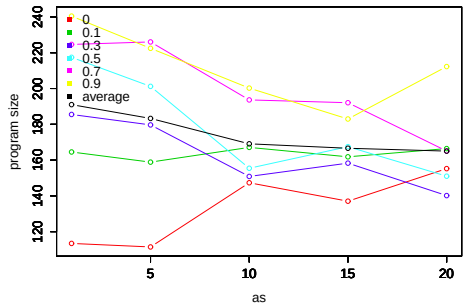


(g) R7（$f(x) = ln(x + 1) + ln(x^2 + 1)$）

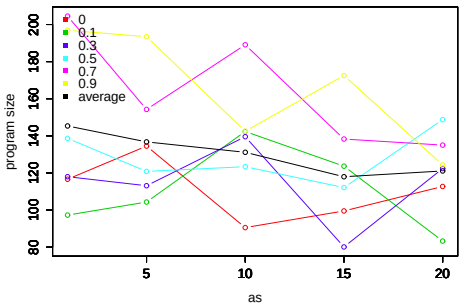


(h) R8（$f(x, y) = sin(x) + sin(y^2)$）

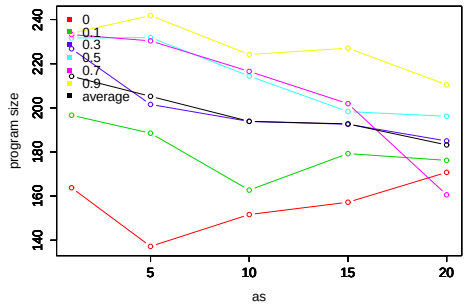
図 8.17　アーカイブサイズ as の変化によるプログラムサイズの変化（Case1）



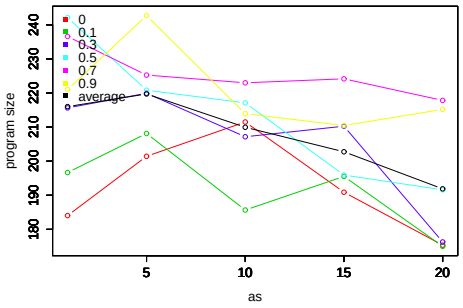
(a) R1（$f(x) = x^4 + x^3 + x^2 + x$）



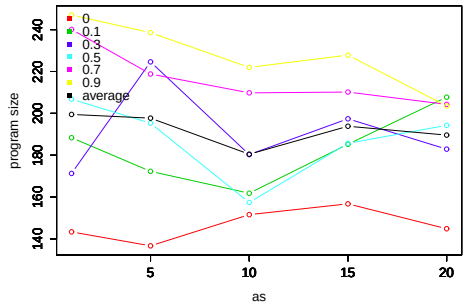
(b) R2（$f(x) = x^5 - 2x^3 + x$）



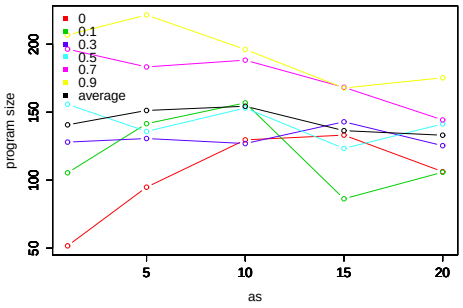
(c) R3（$f(x) = x^6 - 2x^4 + x^2$）



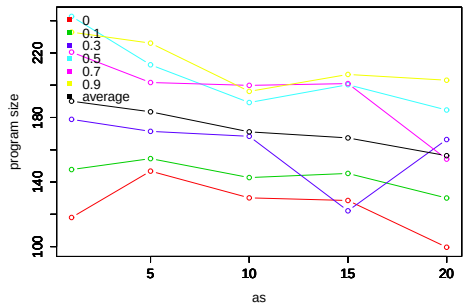
(d) R4（$f(x, y) = x^y$）



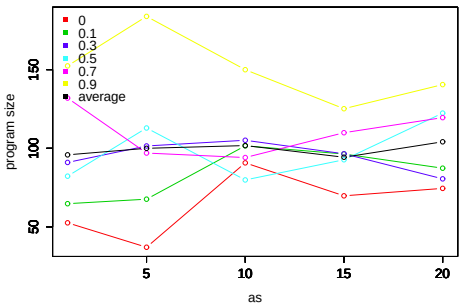
(e) R5（$f(x) = sin(x^2) \times cos(x) - 1$）



(f) R6（$f(x) = sin(x) + sin(x + x^2)$）



(g) R7（$f(x) = ln(x + 1) + ln(x^2 + 1)$）



(h) R8（$f(x, y) = sin(x) + sin(y^2)$）

図 8.18　アーカイブサイズ as の変化によるプログラムサイズの変化（Case4）

$(\mu+\lambda)$-GP と同様に待機時間の上限値を設定する必要がある．適切な上限値が設定できない場合でも，継続的な進化が可能であるため $(\mu+\lambda)$-GP と比べると影響は小さいが，進化の初期において探索性能が低下する．

ARE-GP は，各個体の計算速度にばらつきがある場合，評価が完了しない個体が含まれる場合のいずれにおいても，待機時間の上限値などの特別な設定なしに対応が可能であるという特徴がある．これにより，待機時間の適切な値が未知な問題にも適用可能である．さらに，全個体の評価値を使用可能な同期進化の $(\mu+\lambda)$-GP に比べ短い時間で効率的な解探索が可能であり，かつ適切なパラメータ設定によって従来の非同期進化法である ASSGP よりも優れた解探索性能を示す．ARE-GP と ASSGP の大きな違いは，(1) 親個体の選択方法と (2) 個体削除の方法である．(1) 親個体の選択方法については，ASSGP が評価が完了した 2 個体間のみで親個体を選択するのに対して，ARE-GP が評価が完了した 2 個体にリファレンス個体を加えた 3 個体の中から親個体を選択する．そのため，ASSGP では評価時間が短く評価値も低い 2 個体から親個体が選択される可能性があるのに対し，ARE-GP では優良個体であるリファレンス個体が必ず含まれるため親個体として ASSGP に比べて優れた個体が選択されやすくなる．(2) 個体削除の方法については，ASSGP が親選択の際のトーナメントで敗者となった個体が削除されるのに対して，ARE-GP は親選択の際の適合度削除とリーパーキューの順位に基づくリーパー削除の 2 つの削除機構を持つ．特に適合度削除では，リファレンス個体よりも劣る個体が削除の対象となるため，ASSGP の削除と比べてより厳しい選択圧がかかることになる．．これにより，ARE-GP は ASSGP に比べて優れた解探索性能を示したと考えられる．

ARE-GP と $(\mu+\lambda)$-GP，ASSGP の結果の有意差を統計的に示すために，10^7 単位時間経過後の平均適合度とノンパラメトリック検定の一つであるマン–ホイットニーの U 検定（Mann–Whitney U test） [44] を用いて検定を行った結果の P 値を表 8.13 に示す．検定の結果，$(\mu+\lambda)$-GP との比較ではすべての例題，ケースにおいて有意水準 $\alpha=0.05$ で有意差があることが確認された．また，ASSGP との比較でも，多くの例題とケースで有意水準 $\alpha=0.05$ で有意差があることが確認され，ARE-GP の平均適合度が劣る場合には有意差が見られないことから ASSGP と比較して同等以上の性能を実現できていることが確認された．

また，ARE-GP の特徴として，各個体の計算速度にばらつきがある場合がない場合に比べて収束速度，最終世代の適合度ともに性能が向上する例が確認できる．この特徴を明らかにするために，図 8.19 に Case1 と Case4 の同一経過単位時間における平均適合度の比率を示す．具体的には，同一の手法（ARE-GP，および $(\mu+\lambda)$-GP，ASSGP）において Case1 の経過単位時間 t における平均適合度 $f_{Case1}(t)$ と Case4 における平均適合度 $f_{Case4}(t)$ の比率 $r(t)=f_{Case4}(t)/f_{Case1}(t)$ を示す．図 8.19 において，横軸は経過単位時間，縦軸は対数スケールで $r(t)$ の値を表す．丸プロットは ARE-GP の結果，四

表 8.13　各実験ケースにおける 10^7 単位時間経過後の平均適合度とマン–ホイットニーの U 検定によって得られた P 値（1/2）

		R1	R2	R3	R4
Case1	ARE-GP	$\mathbf{5.9 \times 10^{-3}}$	$\mathbf{1.7 \times 10^{-4}}$	$\mathbf{1.5 \times 10^{-4}}$	$\mathbf{5.9 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	3.6×10^{-2}	8.6×10^{-3}	1.5×10^{-3}	2.8×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	1.3×10^{-2}	6.1×10^{-4}	1.6×10^{-4}	7.2×10^{-3}
	(P 値)	**(0.0072)**	(0.41)	(0.35)	(0.39)
Case2	ARE-GP	$\mathbf{9.1 \times 10^{-3}}$	$\mathbf{1.8 \times 10^{-4}}$	3.6×10^{-4}	$\mathbf{5.1 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	7.7×10^{-2}	6.7×10^{-3}	2.0×10^{-3}	3.0×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	1.0×10^{-2}	1.5×10^{-3}	$\mathbf{1.1 \times 10^{-4}}$	1.4×10^{-2}
	(P 値)	(0.17)	(0.29)	(0.45)	**(< 0.01)**
Case3	ARE-GP	$\mathbf{7.0 \times 10^{-3}}$	$\mathbf{1.8 \times 10^{-4}}$	$\mathbf{1.6 \times 10^{-4}}$	$\mathbf{5.9 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	4.3×10^{-2}	1.1×10^{-2}	1.7×10^{-3}	3.1×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	2.3×10^{-2}	2.3×10^{-3}	2.2×10^{-4}	1.0×10^{-2}
	(P 値)	(0.068)	(0.14)	(0.73)	**(0.013)**
Case4	ARE-GP	$\mathbf{5.4 \times 10^{-3}}$	$\mathbf{7.9 \times 10^{-4}}$	$\mathbf{1.1 \times 10^{-4}}$	$\mathbf{6.6 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	8.0×10^{-2}	1.3×10^{-2}	2.0×10^{-3}	3.7×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	8.0×10^{-3}	1.3×10^{-3}	1.6×10^{-4}	1.1×10^{-2}
	(P 値)	(0.19)	(0.41)	**(0.021)**	**(0.019)**

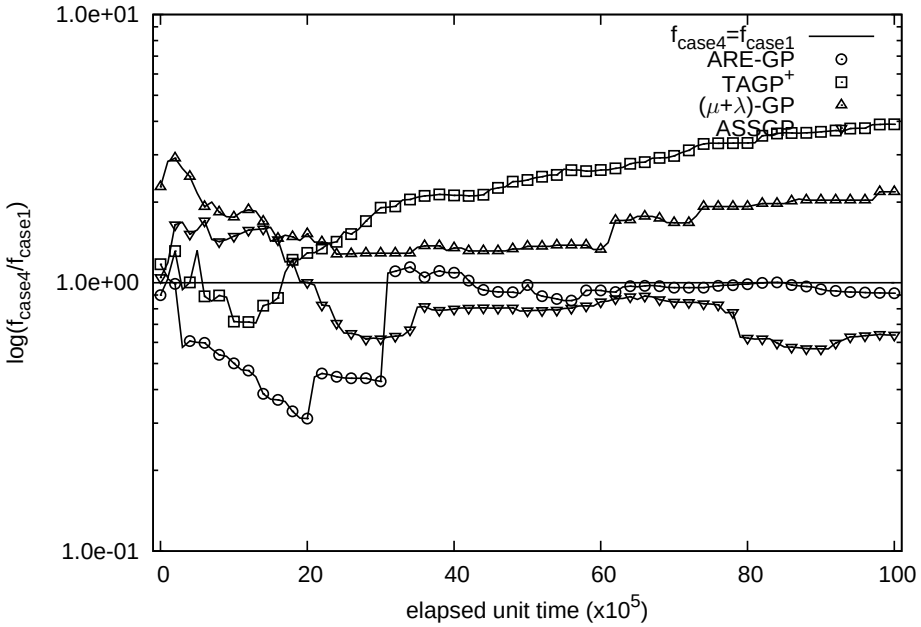
角プロットは TAGP$^+$ の結果，三角プロットは $(\mu+\lambda)$-GP の結果，逆三角プロットは ASSGP の結果を表す．$r(t)$ の値が 1 よりも大きい場合（図 8.19 において 1 よりも上にプロットされる場合），Case1 の平均適合度が Case4 の平均適合度よりも小さいことを意味し，経過単位時間 t において Case1 が Case4 を上回ることを示す．逆に $r(t)$ が 1 よりも小さい場合（図 8.19 において 1 よりも下にプロットされる場合），経過単位時間 t において Case4 が Case1 を上回ることを示す．図 8.19 の結果から，$(\mu+\lambda)$-GP はすべての例題において 1 よりも上にプロットされていることから，Case1 の結果が Case4 の結果を上回っていることがわかる．ASSGP は，R1 と R5 において 1 よりも下にプロットされていることから，Case4 の結果が Case1 の結果を上回っており，R4 と R8 についても 1 付近にプロットされていることから Case4 と Case1 が同等の結果を示していること

表 8.13　各実験ケースにおける 10^7 単位時間経過後の平均適合度とマン–ホイットニーの U 検定によって得られた P 値（2/2）

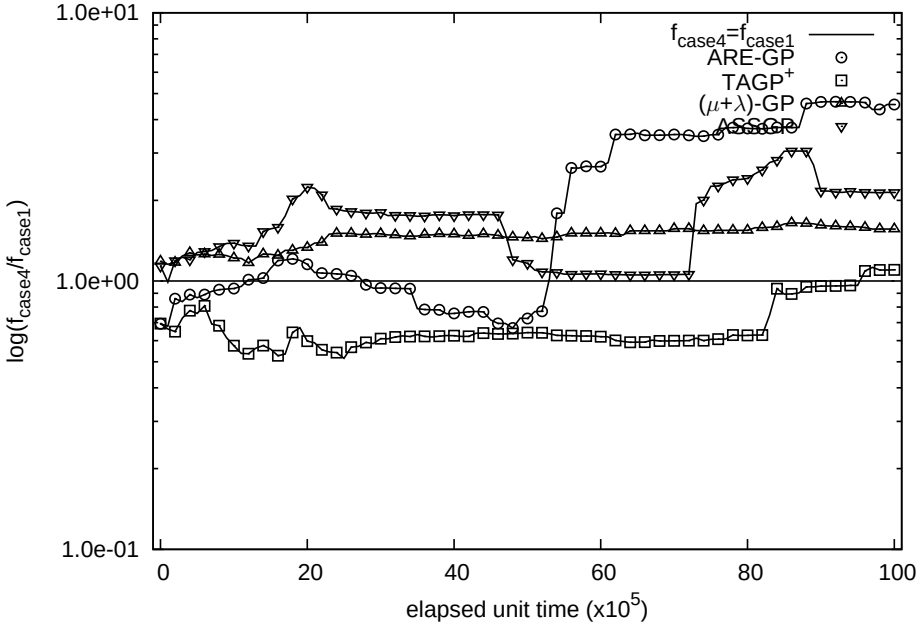
		R5	R6	R7	R8
Case1	ARE-GP	$\mathbf{8.3 \times 10^{-5}}$	1.1×10^{-2}	$\mathbf{1.1 \times 10^{-3}}$	$\mathbf{2.9 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	3.0×10^{-3}	2.9×10^{-2}	7.5×10^{-3}	2.0×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	1.3×10^{-3}	$\mathbf{4.1 \times 10^{-3}}$	1.2×10^{-3}	7.4×10^{-3}
	(P 値)	**(0.021)**	(0.36)	**(0.012)**	**(< 0.01)**
Case2	ARE-GP	$\mathbf{7.7 \times 10^{-5}}$	7.0×10^{-3}	2.8×10^{-3}	$\mathbf{4.6 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	5.2×10^{-3}	2.7×10^{-2}	1.3×10^{-2}	2.3×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	5.0×10^{-4}	$\mathbf{3.6 \times 10^{-3}}$	$\mathbf{1.0 \times 10^{-3}}$	1.1×10^{-2}
	(P 値)	**(< 0.01)**	(0.85)	(0.42)	**(< 0.01)**
Case3	ARE-GP	$\mathbf{3.3 \times 10^{-4}}$	3.848×10^{-3}	$\mathbf{9.5 \times 10^{-4}}$	$\mathbf{4.4 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	2.4×10^{-3}	4.8×10^{-2}	5.6×10^{-3}	1.6×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	5.9×10^{-4}	$\mathbf{3.842 \times 10^{-3}}$	2.9×10^{-3}	1.1×10^{-2}
	(P 値)	(0.39)	(0.31)	(0.92)	**(< 0.01)**
Case4	ARE-GP	$\mathbf{1.4 \times 10^{-4}}$	9.4×10^{-3}	$\mathbf{8.9 \times 10^{-4}}$	$\mathbf{5.5 \times 10^{-3}}$
	$(\mu+\lambda)$-GP	6.1×10^{-3}	4.9×10^{-2}	2.3×10^{-2}	3.2×10^{-2}
	(P 値)	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**	**(< 0.01)**
	ASSGP	3.0×10^{-4}	$\mathbf{6.4 \times 10^{-3}}$	3.2×10^{-3}	8.6×10^{-3}
	(P 値)	(0.29)	(0.84)	**(< 0.01)**	**(0.030)**

がわかる．TAGP$^+$ は，R5 と R6 において 1 よりも下にプロットされていることから，Case4 の結果が Case1 の結果を上回っており，R2 と R3，R8 についても 1 付近にプロットされていることから Case4 と Case1 が同等の結果を示していることがわかる．これに対し，ARE-GP は R1，R3，R6，R7 において 1 よりも下にプロットされていることから，Case4 の結果が Case1 の結果を上回っており，R2 についても 1 付近にプロットされていることから Case4 と Case1 が同等の結果を示していることがわかる．このことから，ARE-GP では他の手法に比べて評価時間にばらつきが大きいほど解探索性能が向上する可能性が示唆された．

GP の評価は個体のプログラムサイズに応じて評価にかかる時間が決定するため，評価が完了した順に親選択を行う ARE-GP において，評価値が一様な環境ではプログラムサ

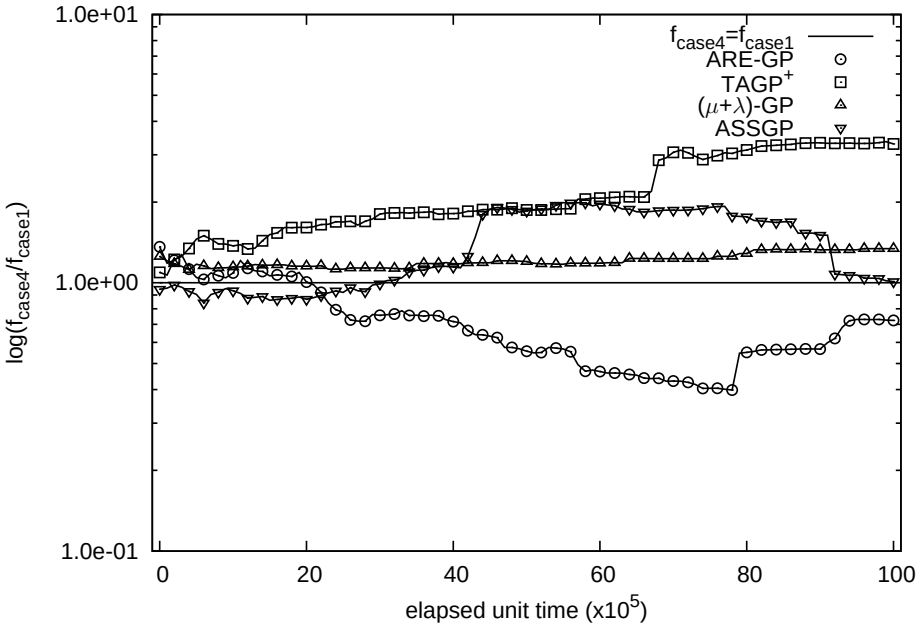


(a) R1 ($f(x) = x^4 + x^3 + x^2 + x$)

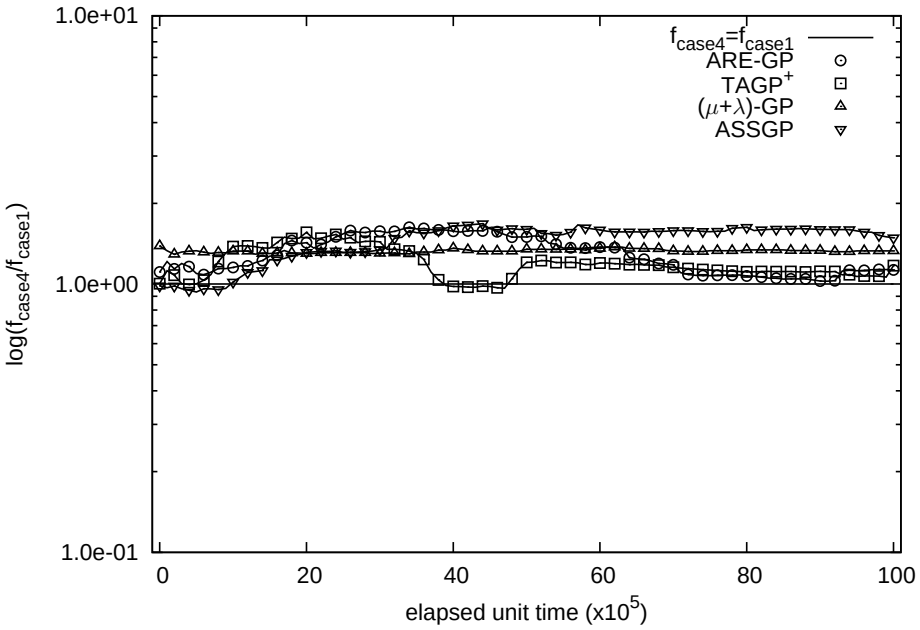


(b) R2 ($f(x) = x^5 - 2x^3 + x$)

図 8.19　Case1 と Case4 の平均適合度の比率の推移（1/4）

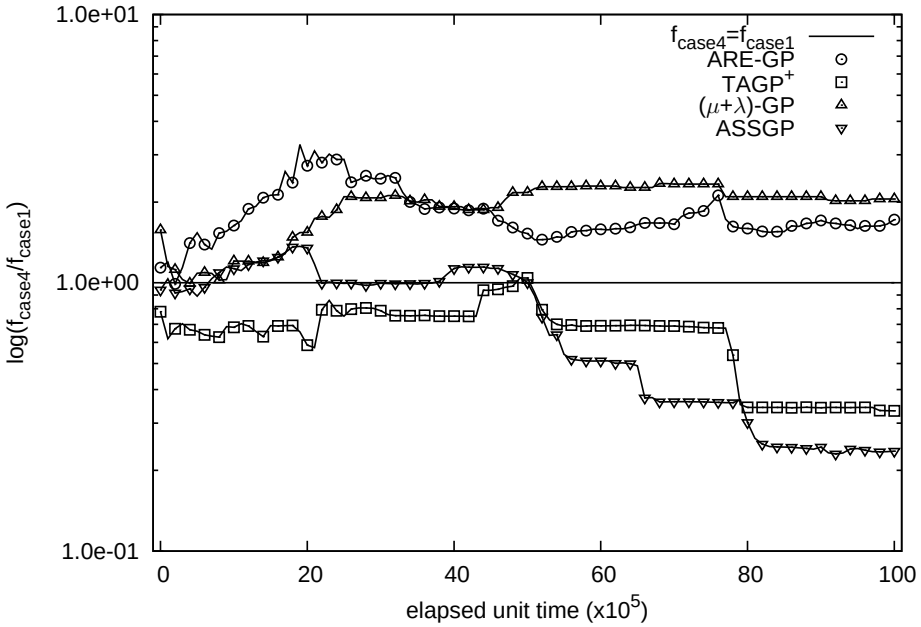


(c) R3（$f(x) = x^6 - 2x^4 + x^2$）

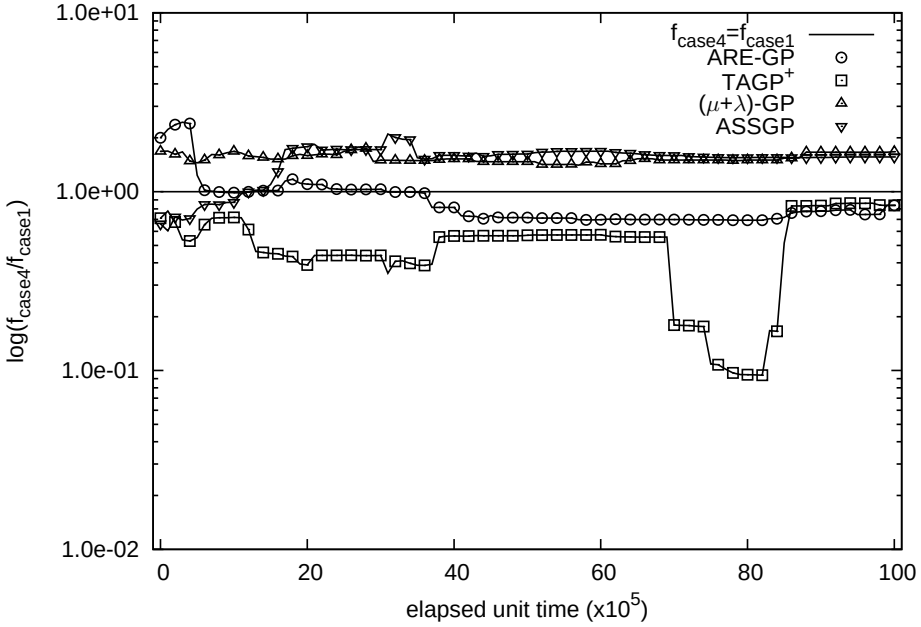


(d) R4（$f(x, y) = x^y$）

図 8.19　Case1 と Case4 の平均適合度の比率の推移（2/4）

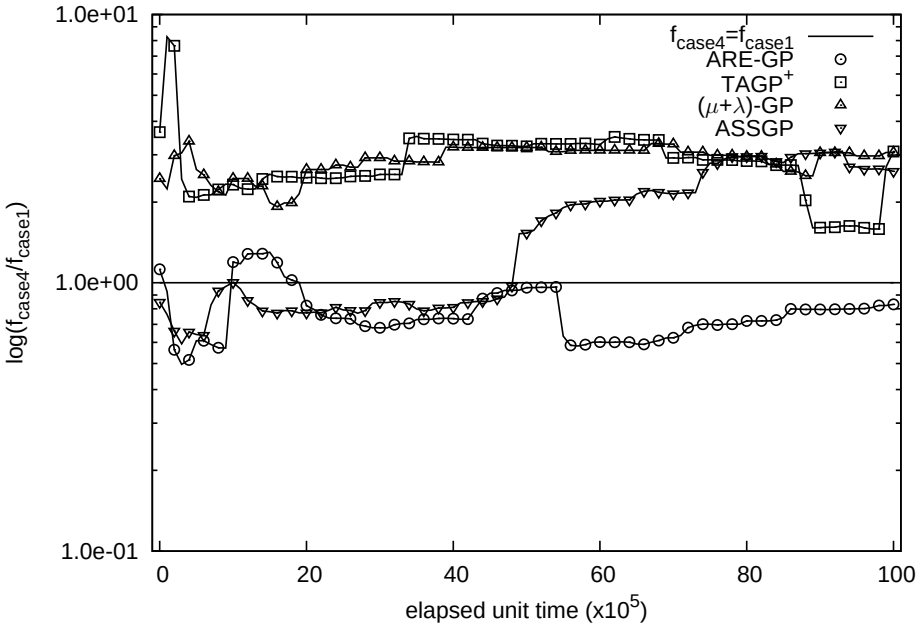


(i) R5 ($f(x) = sin(x^2) \times cos(x) - 1$)

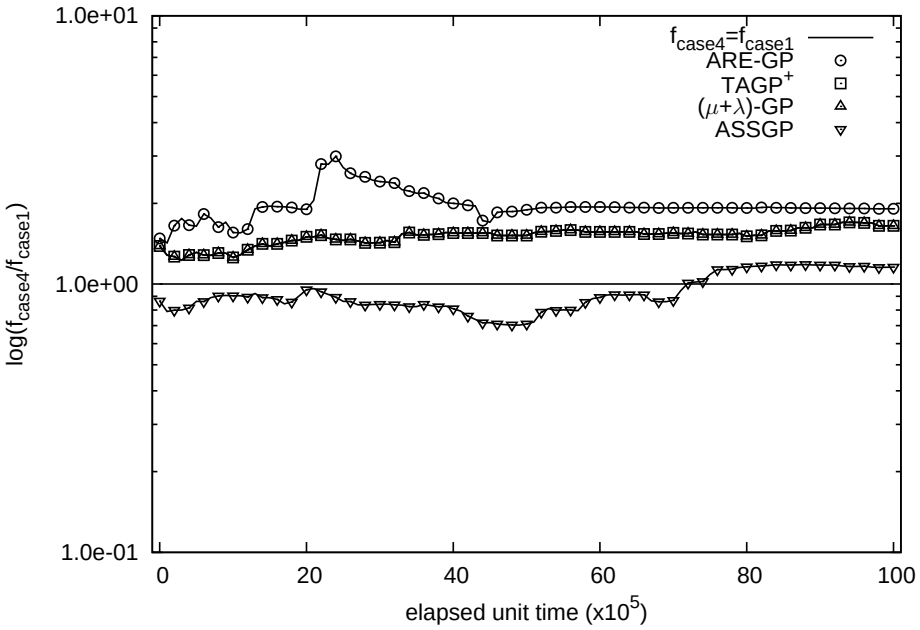


(j) R6 ($f(x) = sin(x) + sin(x + x^2)$)

図 8.19　Case1 と Case4 の平均適合度の比率の推移（3/4）



(g) R7（$f(x) = ln(x+1) + ln(x^2+1)$）



(h) R8（$f(x,y) = sin(x) + sin(y^2)$）

図 8.19　Case1 と Case4 の平均適合度の比率の推移（4/4）

イズが小さい個体が親選択される機会が増加する．しかし計算速度にばらつきが生じることによって，単純にプログラムサイズに応じた順序付けではなくなり，プログラムサイズの大きい個体であっても親選択の機会が増加し，結果として多様な親選択が実現されたと考えられる．これにより，より評価時間のばらつきが大きい Case4 において解の質が向上したと考えられる．

第 9 章

結論

9.1　本研究の成果

本研究では，生物の進化のメカニズムをもとにしたメタヒューリスティックな最適化手法である EA において，解の評価時間が均一でない場合に計算時間の無駄が大きくなるという問題に対し，各解を独立に進化させる非同期 EA に着目した．具体的には，(1) 解を非同期に進化可能であり，(2) 解の評価が完了しない可能性がある場合にも対処可能な新しい非同期 EA を提案し，その有効性を検証することを目的とした．上記目的に向け，本研究では，二種類の非同期 EA を提案した．一つ目は，非同期なデジタル生物の進化をシミュレートする Tierra を拡張し，絶対評価にもとづいて解を進化させる Tierra 型非同期 EA（Tierra-based Asynchronous EA：TAEA）を提案し，進化を促進するために親選択時に過去数個体との比較によって優良個体の選択を促す TD トーナメント選択（temporal difference tournament selection：TDTS）を提案した．さらに，TAEA に適合度関数の自動調整と個体の削除パラーメータを自動的に調節する手法を加えて拡張した適応型 TAEA（TAEA^{+}）を提案した．二つ目は，非同期に得られる部分的な解評価から更新される指標を用いて相対評価に基づいて解を進化させる非同期リファレンス評価を用いる EA（Asynchronous Reference-based Evaluation EA：ARE-EA）を提案した．ARE-EA は，(1) アーカイブによる優良個体の保持と (2) アーカイブから選択したリファレンス個体を用いた相対評価による選択法が全体情報を持たずに最適化できる鍵となっている．

提案手法の有効性を検証するために，(1)GP で一般的なベンチマーク問題である関数同定問題（symbolic regression problem）における関数進化，(2) 実応用を想定した PIC マイコンで実行可能なアセンブリプログラム進化，(3) 宇宙機への適用を念頭においたプログラムのビット反転が発生する環境でのアセンブリプログラム進化に提案手法を適用した．特に，(1) 関数同定問題は，四則演算，三角関数，対数関数，指数関数を用いて与えられた複数のデータ点を通る関数をランダムに生成した関数から進化によって獲得する問

<table>
<tr><th colspan="2">問題の種類
非同期EAの種類</th><th colspan="2">プログラム</th><th rowspan="2">ランダムな関数</th></tr>
<tr><th colspan="2"></th><th>ビット反転なし</th><th>ビット反転あり</th></tr>
<tr><td rowspan="2">評価方法</td><td>絶対評価
TAEA</td><td rowspan="2">実行ステップ数削減割合
SSGP
ASSGP < TAGP+ ≦ ARE-GP
TAGP/TDTS
TAGP</td><td rowspan="2">ビット反転耐性
ARE-GP < TAGP/TDTS</td><td rowspan="2">誤差最小化性能
(μ+λ)-GP < ASSGP ≦ TAGP+ ≦ ARE-GP</td></tr>
<tr><td>相対評価
ARE-EA</td></tr>
</table>

図 9.1　本研究のまとめ

題であり，得られた関数の値と真値との差を最小化することを目的とし，(2) 実応用を想定したアセンブリプログラム進化は，PIC マイコンで実行可能な 33 命令のアセンブリ言語で記述された目的を達成可能なプログラムを与え，その実行ステップ数を最小化することを目的とした．そして，(3) ビット反転が発生する環境でのアセンブリプログラム進化は，宇宙環境において半導体素子に宇宙放射線が衝突することによって情報のビット反転が起こるシングルイベントアップセット（Single-Event Upset）という問題を想定し，アセンブリプログラムの進化中にランダムにプログラムにビット反転が生じる環境下で目的を達成可能なプログラムを維持，進化可能であるかを検証した．

上記の問題に提案手法を適用したところ，次の知見を得た．まず，提案した非同期進化法は，(1) 部分的な解の評価のみで，全体の解の評価を利用可能な従来の同期進化法と同等以上の関数近似を達成可能であり，(2) 解の評価時間のばらつきが大きいほど関数近似性能が向上する．また，(3) 与えられた目的を達成可能なプログラムを維持可能であることに加えて，実行ステップ数の最小化が可能である．さらに，(4) プログラムの改変や変数のエラーが発生してもプログラム進化を可能にすることを明らかにした．詳細は次のようにまとめられる．まず，TAEA を GP に適用した TAGP を用いる実験から，以下の知見を得た．

●**TAEA** の改良による性能向上

1. アセンブリプログラム進化を扱う例題において，TDTS を用いる TAGP（TAGP/TDTS）が TDTS を用いない TAGP に比べて実行ステップ数の少ないプログラムを獲得可能である．
2. TDTS において過去の個体との比較数によって大きな性能差はなく，最小の 1 個体前との比較で十分な進化性能が実現可能である．
3. TAGP/TDTS に四分位数に基づく適合度スケーリング（Quartile based Fitness Scaling：QFS）と適応的リーパー制御パラメータ P^{α}_{down} を加えた TAGP^{+} によって事前の適合度関数の設計なく TAGP/TDTS と同等以上の性能を実現可能である．

●**TAEA** と従来手法の比較

4. TAGP$^+$ がすべての例題において同期 GP の SSGP，$(\mu + \lambda)$-GP，非同期 GP の ASSGP と同等以上の進化性能を示す．獲得されたプログラムを比較すると，SSGP では獲得できない実行ステップ数の短いプログラムが TAGP$^+$ によって生成されていることを確認した．
5. 関数同定問題において，解の評価時間にばらつきがある環境として，(1) 計算速度にばらつきがある場合，(2) 評価の完了しない解がある場合を模擬した実験の結果，(2) 評価の完了しない解がある場合にも TAGP$^+$ は同期 GP の $(\mu + \lambda)$-GP を上回る性能を示し，従来非同期 GP の非同期 steady-state GP（ASSGP）と比較しても同等以上の性能を示した．一方，(1) 計算速度にばらつきがある場合には，一部の例題で TAGP$^+$ が局所解に陥り ASSGP より劣る結果となるものの，その他の例題では ASSGP と比較しても同等以上の性能を示した．

●ビット反転が発生する環境下での **TAEA**

6. ビット反転が発生する環境下でアセンブリプログラムを進化させる実験の結果，TAGP/TDTS が実宇宙では生じないような高いビット反転率においても目的を達成可能なプログラムを維持可能であり，さらに初期に与えたプログラムよりもサイズの小さなプログラムを生成可能である．

次に，ARE-EA を GP に適用した ARE-GP を用いる実験から，以下の知見を得た．

●**ARE-EA** のパラメータによる影響

1. アセンブリプログラム進化を扱う例題，および関数同定問題において ARE-GP に含まれる 2 つのパラメータ（適合度削除確率 P_d とアーカイブサイズ as）についてその影響を分析した結果，適合度削除確率 P_d は高い値（$P_d \geq 0.5$），アーカイブサイズは大きな影響はないものの小さな値（$as \leq 10$）に設定することが望ましい．

●**ARE-EA** と従来手法の比較

2. アセンブリプログラム進化を扱う例題において，ARE-GP がすべての例題で TAGP$^+$，同期 GP の SSGP を上回る性能を示すことが明らかになった．特に，獲得されたプログラムから，ARE-GP は TAGP$^+$ で得られた実行ステップ数が最小のプログラムよりも実行ステップ数を削減したプログラムを生成できていることが確認された．
3. 関数同定問題において，解の評価時間にばらつきがある環境として，(1) 計算速度にばらつきがある場合，(2) 評価の完了しない解がある場合を模擬した実験の結果，解の評価時間にばらつきがある場合，ない場合ともに ARE-GP が同期 GP の $(\mu + \lambda)$-GP を上回る性能を示し，従来非同期 GP の非同期 steady-state GP

（ASSGP）と比較しても同等以上の性能を示した．また，ARE-GP は評価時間にばらつきがある場合の方が評価時間にばらつきがない場合に比べて最適化性能が向上する可能性があることが示唆された．

●ビット反転が発生する環境下での **ARE-EA**

4. ビット反転が発生する環境下でアセンブリプログラムを進化させる実験の結果，ARE-GP も実宇宙では生じないような高いビット反転率においても目的を達成可能なプログラムを維持可能であり，さらに TAGP/TDTS よりもサイズ，実行ステップ数ともに小さなプログラムを生成可能である．しかし，ARE-GP は優良個体を特別に保持するアーカイブを使用しているため，アーカイブに対するビット反転に弱く，極めてビット反転率が高い場合には TAGP/TDTS に比べて目的を達成可能なプログラムを維持できる割合が減少する．

9.2 今後の課題

今後の展開として，本研究で採用した例題以外にも提案手法を適用できるよう適用範囲を広げるため，また，さらなる精度向上のため，以下の課題について取り組む必要がある．まず，TAEA や ARE-EA 固有の課題，およびプログラムを進化させる宇宙機用 OBC の課題について整理し，その後に，全体に関わる課題についてまとめる．

1. TAEA 固有の問題：TAEA$^+$ における QFS と P^{α}_{down} の妥当性

本研究では，TAEA を適合度関数の事前の設計なしに適用可能にするために四分位数に基づく適合度スケーリング（QFS）と多様性維持のためのリーパー制御パラメータ P^{α}_{down} を提案し，これらを加えた適応型 TAEA（TAEA$^+$）を提案した．実験の結果，TAEA$^+$ は従来の非同期 EA である ASSGP と同等以上の性能を有することが確認された一方，特に関数同定問題において計算速度にばらつきがある場合に局所解に陥り性能が低下することが確認された．実験の考察においてその一因として，P^{α}_{down} の欠点を述べ，それを改良した修正版 P^{α}_{down} を用いる実験を通して一定の改善が見られた．このことから，TAEA$^+$ についてはさらなる改善の余地があると考えられる．具体的に，QFS については本研究では最も単純な線形スケーリングをもとにした設計を用いたが，これ以外にも冪関数や複数区間での線形スケーリング等の選択肢も考えられる．また，リーパー制御パラメータについては，修正版 P^{α}_{down} に関してその冪数パラメータの設定を考慮する必要がある．そのため，今後はこれらの設計について TAEA に最適な適合度スケーリング，およびリーパー制御パラメータを検証し，TAEA の性能向上を目指す．

2. ARE-EA 固有の問題：適合度削除確率とアーカイブサイズの設定

ARE-EA は，TAEA，従来の非同期 EA である ASSGP と比べて優れた性能を示した一方，適合度削除確率 P_d とアーカイブサイズ as を適切に設定する必要があった．本研究において，パラメータを変化させた比較の結果として適合度削除確率 P_d の値は高く，アーカイブサイズ as は低く設定することが望ましいという指針を得ることができた．しかし，これらの指針は必ずしもすべての問題に対して成り立つわけではなく，それぞれの指針の中でもパラメータを一意に決定することは困難である．そのため，今後はこれら 2 つのパラメータが問題の特性に応じて進化の性能に与える影響を分析し，問題の特性に応じた詳細なパラメータ設計指針を示すとともに，これらを動的に調整可能な手法を提案し，ARE-EA の汎用性の向上を目指す．

3. プログラムを進化させる宇宙機用 OBC の問題：ビット反転の影響範囲

SEU によるビット反転が生じる環境において TAGP，および ARE-GP が PIC アセンブリプログラムを進化可能であることが明らかになった．しかし，実験ではシステム全体のうちプログラム本体とそれらが使用するレジスタ（変数）領域のビット反転のみを扱い，それ以外の部分，特に GP の実行部に対する影響は考慮していない．そのため，プログラムを進化させる宇宙機用 OBC の実現に向けては，システムのどの部分までビット反転の影響を許容できるかを明確にし，その範囲を拡大することが必須である．具体的には，プログラム集団をメモリ内で管理する部分，プログラム実行部と評価部（評価関数），交叉や突然変異などの遺伝的操作等に対する影響を考慮する必要がある．そして，ビット反転の許容範囲の切り分けに基づいて，ビット反転の影響を許容できる部分についてはビット反転に対する耐性を考慮せず部品を選定でき（例えば，民生部品），ビット反転の影響を許容できない部分についてのみ従来のようなビット反転に耐性のある部品（例えば，FPGA）や金属シールド，多重化などの対策を施すことで，全体として従来よりも少ない SEU 対策でシステムを構成可能になる．

4. 全体に関わる問題

(i) 提案手法の汎用性

本研究では，実験において GP の一部の例題を扱う実験を行ったが，他の GP の例題に適用し，提案手法の有効性の一般性を検証する必要がある．具体的には，GP のベンチマーク問題をまとめた [38] で分類されている論理関数生成や分類問題，人工蟻の採餌行動を獲得する問題などに取り組み，提案手法の有効性と限界を検証する必要がある．また，

評価時間と評価値の関係 / 評価方法	(1)一定	(2)負相関	(3)相関なし	(4)正相関
絶対評価 (TAEA)	一般的な問題	実行ステップ最小化	関数同定問題	ロボット歩行制御
相対評価 (ARE-EA)				

図 9.2　評価時間と評価値の関係性による問題の分類

ベンチマーク問題だけではなく，実問題への適用も検討することが急務である．特に評価時にシミュレーションなどが必要であり，評価時間が遺伝的操作等にかかる時間と比べて十分長く，かつ評価時間が均一でなくなる問題が実問題では想定されることから，このような特性を持つ実問題を対象とした実験がターゲットとなる．

(ii) 提案 EA フレームワークの適用可能性

提案手法は EA のフレームワークであるため，GP のみならず GA など他の EA にも適用可能である．そのため，今後は提案手法を GA をはじめとする他の EA 手法に展開して，その適用可能性を検証する．

(iii) 評価時間と評価値の関係

提案手法では，解の評価が完了するごとに子個体生成，生存選択を実行し，非同期に解を進化させる．しかしこの場合，評価時間の短い解が相対的に子個体生成の際の親として選択される可能性が高くなる．そのため，評価時間と評価値の関係性によって探索性能が左右されることが考えられる．具体的には，図 9.2 に示すように，(1) 評価時間が評価値によらず一定の場合，(2) 評価時間が短いほど評価値が高くなる負の相関関係，(3) 評価時間と評価値の間に相関関係がない，(4) 評価時間が長いほど評価値が高くなる正の相関関係が考えられる．この内，(1) 評価時間が一定の場合は評価時間にばらつきがないため同期 EA で解決可能であり，非同期 EA においても各解に均等に親選択の機会が与えられるため，探索性能には影響を与えない．(2) 負の相関関係については，評価値の高い個体が親として選択される可能性が高くなるため，非同期 EA によって進化が促進される．本研究で扱ったアセンブリプログラムの進化は実行ステップ数（評価時間）が短いほど優れた解であると言えるため負の相関関係を持つ問題に該当する．(3) 相関関係がない場合は，非同期 EA では同一の評価値で評価時間が大きくことなるような場合に評価時間の短い解が優先的に進化するため，より評価時間の短い優良解が獲得される．本研究で扱った関数同定問題は，評価時間にばらつきがあるものの評価時間と評価値の間の相関は小さいため相関関係がない場合に分類される．(4) 正の相関関係については，非同期 EA では評

価時間が短く評価値の低い局所解に収束する可能性があるため最も困難な問題である．正の相関関係を持つ例題として，ロボットの歩行制御最適化において歩行時間が長いほどよい制御であるとする場合などがあげられる．今後は，問題としてこれらの分類を考慮し，特に (4) 正の相関関係を持つ非同期 EA の適用が困難と思われる領域に対して提案手法を改良することが課題となる．

また，それぞれの関係性において評価時間のばらつきの分布の観点からの分析も重要となる．具体的には，各個体間で評価時間の差はどれくらいあるか，それぞれの評価時間を持つ個体は何個体存在するか等を定量的に表現し，それぞれの分布に基づいた分析をする必要がある．

(iv) 評価時間と探索領域の関係

本研究の PIC アセンブリプログラム進化，および関数同定問題において，すべての命令語の実行時間は均一であった．しかし，実際の CPU 上での必要クロック数を考慮すると命令ごとに実行時間に差があると考えられる．例えば，PIC アセンブリプログラムの命令語では，$ADDWF$ や $MOVF$ 等の算術命令は 1 サイクルで実行可能であるのに対して $GOTO$ や $BTFSS$ 等の制御命令は 2 サイクルが必要になる [1]．また，関数同定問題では，$+$，$-$ 等の単純な四則演算に対して sin や ln 等は複雑な演算が必要になるため実行時間は多くなる．このような状況を想定した場合，プログラムごとのプログラムサイズやループの有無以外に使用する命令によっても実行時間（評価時間）に差が生じる．そのため，例えば関数同定問題における R3（$f(x) = sin(x^2) \times cos(x) - 1$）や R4（$f(x) = ln(x+1) + ln(x^2+1)$）の例題のように sin や ln 等の命令が必要になる問題では正しい関数に近いほど評価時間が長くなるため解探索が困難になると考えられる．これは，「(iii) 評価時間と評価値の関係」で述べた目的関数空間上での評価値に依存した評価時間の差に対し，設計変数空間上での探索領域に依存した評価時間の差であるといえる．そのため，今後はこのような探索領域による評価時間差も考慮した検証が課題となる．

(v) 並列計算環境での有効性検証

提案手法を含む非同期 EA は，特に並列計算環境において有効である．本研究では並列計算環境を擬似した実験にとどまっているため，実際の並列計算環境に適用することが重要である．並列計算環境に適用することによって，実計算環境において計算機の性能差や通信環境によって生じる評価時間差による影響を検証出来るだけでなく，遺伝的操作や生存選択におけるオーバーヘッドなど擬似環境では再現できない問題への対応が今後の課題となる．

(vi) 非同期の度合いによる性能の影響

本研究の結果から，非同期 EA が同期 EA を上回る性能を示したが，本研究では非同期の度合いについて極端な例のみを扱っている．具体的には，TDTS を用いない TAEA では 1 個体のみの評価値に基づいて，TDTS を用いる TAEA（TDTS）ではトーナメントサイズ λ に応じた個体数の評価が完了した段階で，ARE-EA では 2 個体の評価が完了した段階でそれぞれ進化をさせるため，完全な非同期，あるいはそれに近い非同期性の EA であるといえる．しかし，この非同期度合いの変更が EA に与える影響，および EA に最も適した非同期性を探求することが急務である．

謝辞

本論文をまとめるにあたり始終多大なるご指導と御教示をいただいた主任指導教員である髙玉圭樹教授，指導教員の吉浦裕教授，西野哲朗教授に心より感謝の意を表します．博士論文の審査をしていただいた高橋治久教授，高橋裕樹准教授，日々の研究生活において適切な助言をいただいた佐藤寛之助教と服部聖彦助教，特別研究員制度によって研究費の面で助成いただいた日本学術振興会様，並びに研究室の人々にこの場を借りてあらためて深く感謝申し上げます．

参考文献

[1] Microchip Technology Inc. *PIC10F200/202/204/206 Data Sheet 6-Pin, 8-bit Flash Microcontrollers*. Microchip Technology Inc., 2007.

[2] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 第 1 版, 1989.

[3] John Koza. *Genetic Programming On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[4] Hans-Paul Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., 1993.

[5] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pp. 1942–1948, 1995.

[6] Rainer Storn and Kenneth Price. Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *J. of Global Optimization*, Vol. 11, No. 4, pp. 341–359, December 1997.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, Vol. 6, No. 2, pp. 182–197, apr 2002.

[8] Qingfu Zhang and Hui Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Trans. Evolutionary Computation*, Vol. 11, No. 6, pp. 712–731, 2007.

[9] M. Ueno, S. Usui, H. Tanaka, and A. Watanabe. Technological overview of the next generation Shinkansen high-speed train Series N700. 2008.

[10] Shigeru Obayashi, Shinkyu Jeong, Koji Shimoyama, Kazuhisa Chiba, and Hiroyuki Morino. Multi-Objective Design Exploration and its Applications. *International Journal of Aeronautical and Space Sciences*, Vol. 4, No. 4, Dec 2010.

[11] Christina Bonnington. Teen's iOS App Uses Complex Algorithms to Sum-

marize the Web. http://www.wired.com/2011/12/summly-app-summarization/, Dec 2011.

[12] A Chipperfield and P Fleming. Parallel genetic algorithms. *Parallel and distributed computing handbook*, pp. 1118–1143, 1996.

[13] D.K. Tasoulis, N.G. Pavlidis, V.P. Plagianakos, and M.N. Vrahatis. Parallel differential evolution. In *Evolutionary Computation, 2004. CEC2004. Congress on*, Vol. 2, pp. 2023 – 2029 Vol.2, june 2004.

[14] Jui-Fang Chang, Shu-Chuan Chu, John F. Roddick, and Jeng-Shyang Pan. A parallel particle swarm optimization algorithm with communication strategies. *Journal of Information Science and Engineering*, pp. 809–818, 2005.

[15] Juan J. Durillo, Qingfu Zhang, Antonio J. Nebro, and Enrique Alba. Distribution of Computational Effort in Parallel MOEA/D. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, Vol. 6683 of *Lecture Notes in Computer Science*, pp. 488–502. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[16] Andrew Lewis, Sanaz Mostaghim, and Ian Scriven. Asynchronous multi-objective optimisation in unreliable distributed environments. In Andrew Lewis, Sanaz Mostaghim, and Marcus Randall, editors, *Biologically-Inspired Optimisation Methods*, Vol. 210 of *Studies in Computational Intelligence*, pp. 51–78. Springer Berlin Heidelberg, 2009.

[17] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Vol. 1, pp. 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[18] A. Carlisle and G. Dozier. An off-the-shelf PSO. In *PSO Workshop*. Indianapolis, IN, April 2001.

[19] Byung il Koh, Alan D. George, Raphael T. Haftka, and Benjamin J. Fregly. Parallel asynchronous particle swarm optimization. In *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING 67(4)*, pp. 578–595, 2006.

[20] J. Atienza, M. Garca, J. Gonzlez, and J.J. Merelo. Jinetic: a distributed, fine-grained, asynchronous evolutionary algorithm using jini, 2000.

[21] Tobias Glasmachers. A natural evolution strategy with asynchronous strategy updates. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, GECCO '13, pp. 431–438, New York, NY, USA, 2013. ACM.

[22] Thomas S. Ray. An approach to the synthesis of life. *Artificial Life II*, Vol. XI,

pp. 371–408, 1991.

[23] Markus Brameier and Wolfgang Banzhaf. A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, Vol. 5, pp. 17–26, 2000.

[24] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[25] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[26] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, Vol. 2, pp. 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[27] Advances in genetic programming. chapter A Compiling Genetic Programming System That Directly Manipulates the Machine Code, pp. 311–331. MIT Press, Cambridge, MA, USA, 1994.

[28] Peter Nordin and Wolfgang Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In Larry J. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pp. 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[29] J. C. Bear. Basic concepts in population, quantitative, and evolutionary genetics. *Cell*, Vol. 48, No. 1, p. 9, 2014/10/19.

[30] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In J.D. Becker, I. Eisele, and F.W. Mündemann, editors, *Parallelism, Learning, Evolution*, Vol. 565 of *Lecture Notes in Computer Science*, pp. 398–406. Springer Berlin Heidelberg, 1991.

[31] Bernard Manderick and Piet Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 428–433, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[32] Craig W. Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In *Proc. 2nd International Conf. on Simulation of Adaptive Behavior*, pp. 384–392. MIT Press, 1993.

[33] Thomas S. Ray. Documentation for the Tierra Simulator. http://life.ou.edu/pubs/doc/index.html, 1991.

[34] 木目沢司, T.S.Ray. 人工生命システム Tierra. テクニカルレポート TR-H-268, ATR 人間情報通信研究所, 1999.

[35] ATR 進化システム研究室. 人工生命と進化システム. 東京電機大学出版局, 1998.

[36] C. G. Langton. *Artificial Life*. Addison-Wesley, 1989.

[37] De Jong and Kenneth Alan. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, Department of Computer and Communications Sciences, University of Michigan, 1975.

[38] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth A. De Jong, and Una-May O'Reilly. Genetic Programming Needs Better Benchmarks. In *GECCO*, pp. 791–798, 2012.

[39] リード・ビジネス・インフォメーション株式会社. EDN Japan 2 月号. *EDN Japan*, 2005.

[40] G.M. Swift and S.M. Guertin. In-flight observations of multiple-bit upset in DRAMs. *Nuclear Science, IEEE Transactions on*, Vol. 47, No. 6, pp. 2386–2391, dec 2000.

[41] 池田直美, 新藤浩之, 飯出芳弥, 浅井弘彰, 久保山智司, 松田純夫. MDS-1(つばさ) 搭載民生用メモリ素子のシングルイベント効果に関する解析結果. 民生部品・コンポーネント実証衛星つばさ：MDS-1 軌道上実証成果報告書, 宇宙航空研究開発機構, 2004.

[42] Naomi IKEDA, Hiroyuki SHINDOU, Yoshiya IIDE, Hiroaki ASAI, Satoshi KUBOYAMA, and Sumio MATSUDA. Evaluation of the Errors of Commercial Semiconductor Devices in a Space Radiation Environment. *The transactions of the Institute of Electronics, Information and Communication Engineers. B*, Vol. 88, No. 1, pp. 108–116, 2005.

[43] J. Justesen and T. Hoholdt. *A Course In Error-Correcting Codes*. European Mathematical Society, 2004.

[44] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, Vol. 18, No. 1, pp. 50–60, 03 1947.

関連論文の印刷公表の方法および時期

1. 全著者名：Tomohiro Harada, Masayuki Otani, Yoshihiro Ichikawa, Kiyohiko Hattori, Hiroyuki Sato, Keiki Takadama
   論文題目：Robustness to Bit Inversion in Registers and Acceleration of Program Evolution in On-Board Computer
   印刷公表の方法および時期：Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII), Vol.15, No.8, 2011
   (第 4 章，第 6.3 章に関連)
2. 全著者名：Tomohiro Harada, Keiki Takadama
   論文題目：Asynchronously Evolving Solutions with Excessively Different Evaluation Time by Reference-based Evaluation
   印刷公表の方法および時期：Genetic and Evolutionary Computation Conference 2014 (GECCO2014), pp. 911-918, Vancouver, Canada, Jul, 2014
   (第 5 章，第 6.1 章，第 8.4 章に関連)
3. 全著者名：Tomohiro Harada, Keiki Takadama
   論文題目：Maintaining, Minimizing, and Recovering Machine Language Program through SEU in On-Board Computer
   印刷公表の方法および時期： International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS2014), Montreal, Canada, Jun, 2014
   (第 4 章，第 6.3 章，第 7.3 章に関連)
4. 全著者名：Tomohiro Harada, Keiki Takadama
   論文題目：Asynchronous Evolution by Reference-based Evaluation: Tertiary Parent Selection and its Archive
   印刷公表の方法および時期：17th European Conference on Genetic Programming (EuroGP2014), Lecture Notes in Computer Science, Vol. 8599, Springer-Verlag,

pp. 198-209, Granada, Spain, Apr, 2014
(第 5 章，第 6.2 章，第 8.2 章に関連)

5. 全著者名：Tomohiro Harada, Keiki Takadama
論文題目：Analyzing Program Evolution in Genetic Programming using Asynchronous Evaluation
印刷公表の方法および時期：In Pietro Liò, Orazio Miglino, Giuseppe Nicosia, Stefano Nolfi and Mario Pavone editors, Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems, ECAL 2013, pp. 713-720, Taormina, Italy, Sep, 2013
(第 4 章，第 7.3 章に関連)
6. 全著者名：Tomohiro Harada, Keiki Takadama
論文題目：Asynchronous Evaluation based Genetic Programming: Comparison of Asynchronous and Synchronous Evaluation and its Analysis
印刷公表の方法および時期：16th European Conference on Genetic Programming (EuroGP2013), Lecture Notes in Computer Science, Vol. 7831, Springer-Verlag, pp. 241-252, Apr, 2013
(第 4 章，第 6.2 章，第 7.2 章に関連)
7. 全著者名：Tomohiro Harada, Yoshihiro Ichikawa, Keiki Takadama
論文題目：Evolving Conditional Branch Program in Tierra-based Asynchronous Genetic Programming
印刷公表の方法および時期：The 6th International Conference on Soft Computing and Intelligent Systems, and the 13th International Symposium on Advanced Intelligent Systems (SCIS-ISIS2012), Sep, 2012
(第 4 章，第 6.2 章に関連)
8. 全著者名：Tomohiro Harada, Keiki Takadama
論文題目：Adaptive Mutation Depending on Program Size in Asynchronous Program Evolution
印刷公表の方法および時期： The Third World Congress on Nature and Biologically Inspired Computing (NaBIC2011), Oct, 2011
(第 4 章，第 6.3 章に関連)
9. 全著者名：Tomohiro Harada, Masayuki Otani, Hiroyasu Matsushima, Kiyohiko Hattori, Keiki Takadama
論文題目：Evolving Complex Programs in Tierra-based On-Board Computer on UNITEC-1
印刷公表の方法および時期：The 61st International Astronautical Congress

(IAC2010), Sep, 2010
(第 4 章，第 6.3 章に関連)

10. 全著者名：Tomohiro Harada, Masayuki Otani, Yoshihiro Ichikawa, Kiyohiko Hattori, Hiroyuki Sato, Keiki Takadama
論文題目：Robustness to Bit Inversion in Registers and Acceleration of Program Evolution in On-Board Computer
印刷公表の方法および時期：The 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS2010), Sep, 2010
(第 4 章，第 6.3 章に関連)

11. 全著者名：原田智広，髙玉圭樹
論文題目：非同期評価に基づく遺伝的プログラミングによる機械語プログラムの進化
印刷公表の方法および時期：計測自動制御学会，システム・情報部門，第 7 回関係論的システム科学調査研究会，Jul，2014
(第 4 章，第 5 章，第 6.2 章，第 7.2 章，第 8.2 章に関連)

12. 全著者名：原田智広，髙玉圭樹
論文題目：第三の親個体とそのアーカイブを用いたリファレンス評価による非同期進化
印刷公表の方法および時期：第 6 回進化計算研究会，進化計算学会，pp. 23-32，Mar，2014
(第 5 章，第 6.1 章，第 8.4 章に関連)

13. 全著者名：原田智広，髙玉圭樹
論文題目：リファレンス評価による非同期進化：第三の親選択とそのアーカイブ
印刷公表の方法および時期：進化計算シンポジウム 2013，進化計算学会，pp. 362-369，Dec，2013
(第 5 章，第 6.1 章，第 6.2 章，第 8.2 章，第 8.4 章に関連)

14. 全著者名：原田智広，髙玉圭樹
論文題目：非同期評価型遺伝的プログラミングによる性能向上と同期評価との比較
印刷公表の方法および時期：進化計算シンポジウム 2012，pp. 167-174，Dec，2012
(第 4 章，第 6.2 章，第 7.2 章に関連)

15. 全著者名：原田智広，大谷雅之，市川嘉裕，服部聖彦，佐藤寛之，髙玉圭樹
論文題目：Tierra 型オンボードコンピュータにおけるマルチビットアップセットへの耐性
印刷公表の方法および時期：第 55 回宇宙科学技術連合講演会，Nov，2011
(第 4 章，第 6.3 章に関連)

16. 全著者名：原田智広，大谷雅之，松島裕康，服部聖彦，佐藤寛之，髙玉圭樹
論文題目：Tierra 型非同期 GA：プログラム進化と維持
印刷公表の方法および時期：進化計算シンポジウム 2009，Dec，2009
（第 4 章，第 6.3 章に関連）
17. 全著者名：原田智広，大谷雅之，松島裕康，服部聖彦，髙玉圭樹
論文題目：Tierra 型宇宙機 CPU における耐ビット反転とプログラム進化
印刷公表の方法および時期：第 53 回宇宙科学技術連合講演会，Sep，2009
（第 4 章，第 6.3 章に関連）

付録

A　略語一覧

表 A.1　略語一覧（1/2）

略語	非省略形
EA	進化的アルゴリズム．Evolutionary Algorithm.
GA	遺伝的アルゴリズム．Genetic Algorithm.
GP	遺伝的プログラミング．Genetic Programming.
ES	進化戦略．Evolutionary Strategy.
PSO	粒子群最適化．Particle Swarm Optimization.
DE	差分進化．Differential Evolution.
MOEA	多目的進化的アルゴリズム．Multi-Objective Evolutionary Algorithm.
NSGA-II	非優越ソートを用いる遺伝的アルゴリズム． Non-dominated Sorting GA-II.
MOEA/D	集約関数を用いる多目的進化的アルゴリズム．MOEA on decomposition.
PEA	並列型進化的アルゴリズム．Parallel EA.
TAEA	Tierra 型非同期進化的アルゴリズム．Tierra-based Asynchronous EA.
TAGP	Tierra 型非同期遺伝的プログラミング．Tierra-based Asynchronous GP.
TDTS	時間差分トーナメント選択．temporal difference tournament selection.
ARE-EA	非同期リファレンス評価を用いる進化的アルゴリズム． Asynchronous Reference-based Evaluation EA.
ARE-GP	非同期リファレンス評価を用いる遺伝的プログラミング． Asynchronous Reference-based Evaluation GP.
SSGA	steady-state GA.
SSGP	steady-state GP.
ASSGP	非同期 SSGP．asynchronous SSGP.

表 A.1　略語一覧（2/2）

略語	非省略形
TGP	木構造を用いる GP．Tree GP.
LGP	配列構造を用いる GP．Linear GP.
CGP	Cartesian GP.

B　アセンブリプログラム進化問題における各例題の初期プログラム

以下に，アセンブリプログラム進化問題における各例題の初期プログラムを示す．各図において，“//” 以降は各命令の操作を説明するコメントを表す．

```
 1: CLRF     R2 1// R2 <- 0
 2: MOVF     R1 0// W <- R1
 3: MOVWF    R5 1// R5 <- W
 4: MOVWF    R6 1// R6 <- W
 5: MOVF     R6 0// W <- R6
 6: MOVWF    R7 1// R7 <- W
 7: MOVLW    32   // W <- 32
 8: MOVWF    R6 1// R6 <- 8
 9: LABEL    0    // label(0)
10: MOVF     R5 0// W <- R5
11: BCF      STATUS 0
12: BTFSC    R7 0// if R7[0] == 0 then skip
13: ADDWF    R2 1// R2 <- R2 + W
14: RRF      R2 1// R2 <- R2 >> 1
15: RRF      R7 1// R7 <- R7 >> 1
16: DECFSZ   R6 1// R6 <- R6 - 1
                 // and if R6 == 0 then skip
17: GOTO     0    // goto label(0)
18: CLRF     R3 1// R3 <- 0
19: MOVF     R1 0// W <- R1
20: MOVWF    R5 1// R5 <- W
21: MOVF     R2 0// W <- R2
22: MOVWF    R6 1// R6 <- W
23: MOVF     R6 0// W <- R6
24: MOVWF    R7 1// R7 <- W
25: MOVLW    32   // W <- 32
26: MOVWF    R6 1// R6 <- 8
27: LABEL    1    // label(1)
28: MOVF     R5 0// W <- R5
29: BCF      STATUS 0
30: BTFSC    R7 0// if R7[0] == 0 then skip
31: ADDWF    R3 1// R3 <- R3 + W
32: RRF      R3 1// R3 <- R3 >> 1
33: RRF      R7 1// R7 <- R7 >> 1
34: DECFSZ   R6 1// R6 <- R6 - 1
                 // and if R6 == 0 then skip
35: GOTO     1    // goto label(1)
36: CLRF     R4 1// R4 <- 0
37: MOVF     R1 0// W <- R1
38: MOVWF    R5 1// R5 <- W
39: MOVF     R3 0// W <- R3
40: MOVWF    R6 1// R6 <- W
41: MOVF     R6 0// W <- R6
42: MOVWF    R7 1// R7 <- W
43: MOVLW    32   // W <- 32
44: MOVWF    R6 1// R6 <- 8
45: LABEL    2    // label(2)
46: MOVF     R5 0// W <- R5
47: BCF      STATUS 0
48: BTFSC    R7 0// if R7[0] == 0 then skip
49: ADDWF    R4 1// R4 <- R4 + W
50: RRF      R4 1// R4 <- R4 >> 1
51: RRF      R7 1// R7 <- R7 >> 1
52: DECFSZ   R6 1// R6 <- R6 - 1
                 // and if R6 == 0 then skip
53: GOTO     2    // goto label(2)
54: CLRF     R0 1// R0 <- 0
55: MOVF     R1 0// W <- R1
56: ADDWF    R0 1// R0 <- R0 + W
57: MOVF     R2 0// W <- R2
58: ADDWF    R0 1// R0 <- R0 + W
59: MOVF     R3 0// W <- R3
60: ADDWF    R0 1// R0 <- R0 + W
61: MOVF     R4 0// W <- R4
62: ADDWF    R0 1// R0 <- R0 + W
```

図 B.1　A1 ($f(x) = x^4 + x^3 + x^2 + x$) の初期プログラム

```
 1: CLRF     R2 1// R2 <- 0
 2: MOVF     R1 0// W <- R1
 3: MOVWF    R5 1// R5 <- W
 4: MOVWF    R6 1// R6 <- W
 5: MOVF     R6 0// W <- R6
 6: MOVWF    R7 1// R7 <- W
 7: MOVLW    32   // W <- 32
 8: MOVWF    R6 1// R6 <- 8
 9: LABEL    0    // label(0)
10: MOVF     R5 0// W <- R5
11: BCF      STATUS 0
12: BTFSC    R7 0// if R7[0] == 0 then skip
13: ADDWF    R2 1// R2 <- R2 + W
14: RRF      R2 1// R2 <- R2 >> 1
15: RRF      R7 1// R7 <- R7 >> 1
16: DECFSZ   R6 1// R6 <- R6 - 1
                  // and if R6 == 0 then skip
17: GOTO     0    // goto label(0)
18: CLRF     R3 1// R3 <- 0
19: MOVF     R1 0// W <- R1
20: MOVWF    R5 1// R5 <- W
21: MOVF     R2 0// W <- R2
22: MOVWF    R6 1// R6 <- W
23: MOVF     R6 0// W <- R6
24: MOVWF    R7 1// R7 <- W
25: MOVLW    32   // W <- 32
26: MOVWF    R6 1// R6 <- 8
27: LABEL    1    // label(1)
28: MOVF     R5 0// W <- R5
29: BCF      STATUS 0
30: BTFSC    R7 0// if R7[0] == 0 then skip
31: ADDWF    R3 1// R3 <- R3 + W
32: RRF      R3 1// R3 <- R3 >> 1
33: RRF      R7 1// R7 <- R7 >> 1
34: DECFSZ   R6 1// R6 <- R6 - 1
                  // and if R6 == 0 then skip
35: GOTO     1    // goto label(1)
36: CLRF     R4 1// R4 <- 0
37: MOVF     R2 0// W <- R2
38: MOVWF    R5 1// R5 <- W
39: MOVF     R3 0// W <- R3
40: MOVWF    R6 1// R6 <- W
41: MOVF     R6 0// W <- R6
42: MOVWF    R7 1// R7 <- W
43: MOVLW    32   // W <- 32
44: MOVWF    R6 1// R6 <- 8
45: LABEL    2    // label(2)
46: MOVF     R5 0// W <- R5
47: BCF      STATUS 0
48: BTFSC    R7 0// if R7[0] == 0 then skip
49: ADDWF    R4 1// R4 <- R4 + W
50: RRF      R4 1// R4 <- R4 >> 1
51: RRF      R7 1// R7 <- R7 >> 1
52: DECFSZ   R6 1// R6 <- R6 - 1
                  // and if R6 == 0 then skip
53: GOTO     2    // goto label(2)
54: CLRF     R0 1// R0 <- 0
55: MOVF     R1 0// W <- R1
56: ADDWF    R0 1// R0 <- R0 + W
57: MOVF     R4 0// W <- R4
58: ADDWF    R0 1// R0 <- R0 + W
59: MOVF     R3 0// W <- R3
60: SUBWF    R0 1// R0 <- R0 - W
61: SUBWF    R0 1// R0 <- R0 - W
```

図 B.2　A2 ($f(x) = x^5 - 2x^3 + x$) の初期プログラム

```
 1: CLRF     R2 1// R2 <- 0
 2: MOVF     R1 0// W <- R1
 3: MOVWF    R5 1// R5 <- W
 4: MOVWF    R6 1// R6 <- W
 5: MOVF     R6 0// W <- R6
 6: MOVWF    R7 1// R7 <- W
 7: MOVLW    32  // W <- 32
 8: MOVWF    R6 1// R6 <- 8
 9: LABEL    0   // label(0)
10: MOVF     R5 0// W <- R5
11: BCF      STATUS 0
12: BTFSC    R7 0// if R7[0] == 0 then skip
13: ADDWF    R2 1// R2 <- R2 + W
14: RRF      R2 1// R2 <- R2 >> 1
15: RRF      R7 1// R7 <- R7 >> 1
16: DECFSZ   R6 1// R6 <- R6 - 1
                 // and if R6 == 0 then skip
17: GOTO     0   // goto label(0)
18: CLRF     R3 1// R3 <- 0
19: MOVF     R2 0// W <- R2
20: MOVWF    R5 1// R5 <- W
21: MOVF     R2 0// W <- R2
22: MOVWF    R6 1// R6 <- W
23: MOVF     R6 0// W <- R6
24: MOVWF    R7 1// R7 <- W
25: MOVLW    32  // W <- 32
26: MOVWF    R6 1// R6 <- 8
27: LABEL    1   // label(1)
28: MOVF     R5 0// W <- R5
29: BCF      STATUS 0
30: BTFSC    R7 0// if R7[0] == 0 then skip
31: ADDWF    R3 1// R3 <- R3 + W
32: RRF      R3 1// R3 <- R3 >> 1
33: RRF      R7 1// R7 <- R7 >> 1
34: DECFSZ   R6 1// R6 <- R6 - 1
                 // and if R6 == 0 then skip
35: GOTO     1   // goto label(1)
36: CLRF     R4 1// R4 <- 0
37: MOVF     R2 0// W <- R2
38: MOVWF    R5 1// R5 <- W
39: MOVF     R3 0// W <- R3
40: MOVWF    R6 1// R6 <- W
41: MOVF     R6 0// W <- R6
42: MOVWF    R7 1// R7 <- W
43: MOVLW    32  // W <- 32
44: MOVWF    R6 1// R6 <- 8
45: LABEL    2   // label(2)
46: MOVF     R5 0// W <- R5
47: BCF      STATUS 0
48: BTFSC    R7 0// if R7[0] == 0 then skip
49: ADDWF    R4 1// R4 <- R4 + W
50: RRF      R4 1// R4 <- R4 >> 1
51: RRF      R7 1// R7 <- R7 >> 1
52: DECFSZ   R6 1// R6 <- R6 - 1
                 // and if R6 == 0 then skip
53: GOTO     2   // goto label(2)
54: CLRF     R0 1// R0 <- 0
55: MOVF     R2 0// W <- R2
56: ADDWF    R0 1// R0 <- R0 + W
57: MOVF     R4 0// W <- R4
58: ADDWF    R0 1// R0 <- R0 + W
59: MOVF     R3 0// W <- R3
60: SUBWF    R0 1// R0 <- R0 + W
61: SUBWF    R0 1// R0 <- R0 + W
```

図 B.3　A3 ($f(x) = x^6 - 2x^4 + x^2$) の初期プログラム

```
 1: MOVLW    1        // W<-1
 2: MOVWF    R0 1     // R0<-W
 3: MOVF     R2 1     // R2<-R2 (zero check)
 4: BTFSC    STATUS 2 // if !(STATUS & 0x0004)
 5: GOTO     2        // goto label(2)
 6: NOP      R1 0     // label 17
 7: MOVF     R0 0     // W <- R0
 8: MOVWF    R3 1     // R3 <- W
 9: CLRF     R0 1     // R0 <- 0
10: MOVF     R3 0     // W <- R3
11: MOVWF    R4 1     // R4 <- W
12: MOVLW    32       // W <- 32
13: MOVWF    R3 1     // R3 <- W
14: LABEL    0        // label(0)
15: MOVF     R1 0     // W <- R1
16: BCF      STATUS 0
17: BTFSC    R4 0     // if R4[0] == 0 then skip next inst.
18: ADDWF    R0 1     // R0 <- R0 + W
19: RRF      R0 1     // R0 <- R0 >> 1
20: RRF      R4 1     // R4 <- R4 >> 1
21: DECFSZ   R3 1     // R3 <- R3 - 1
                      // and if R3 == 0 then skip next inst.
22: GOTO     0        // goto label(0)
23: DECFSZ   R2 1     // R2 <- R2 - 1
                      // and if R2 == 0 then skip next inst
24: GOTO     1        // goto label(1)
25: LABEL    2        // label(2)
```

図 B.4　A4 ($f(x, y) = x^y$) の初期プログラム

```
 1: CLRF     R9 1// R9<-0
 2: MOVF     R1 0// W<-R1
 3: ADDWF    R9 1// R9<-R9+W
 4: MOVF     R2 0// W<-R2
 5: ADDWF    R9 1// R9<-R9+W
 6: MOVF     R3 0// W<-R3
 7: ADDWF    R9 1// R9<-R9+W
 8: MOVF     R4 0// W<-R4
 9: ADDWF    R9 1// R9<-R9+W
10: MOVF     R5 0// W<-R5
11: ADDWF    R9 1// R9<-R9+W
12: MOVF     R6 0// W<-R6
13: ADDWF    R9 1// R9<-R9+W
14: MOVF     R7 0// W<-R7
15: ADDWF    R9 1// R9<-R9+W
16: MOVF     R8 0// W<-R8
17: ADDWF    R9 1// R9<-R9+W
18: MOVLW    0   // W<-0
19: BTFSC    R9 0// if R9 & 0x0001
20: MOVLW    1   // W<-1
21: MOVWF    R0 1// R0<-W
```

図 B.5　B1 (8bit-Parity) の初期プログラム

```
 1: MOVF     R7 0     // W<-R7
 2: MOVWF    R11 1    // R11<-W
 3: MOVF     R1 0     // W<-R1
 4: XORWF    R4 0     // W<-R4^W
 5: XORWF    R11 0    // W<-R11^W
 6: MOVWF    R8 1     // R8<-W
 7: MOVF     R1 0     // W<-R1
 8: ANDWF    R4 0     // W<-R4&W
 9: MOVWF    R0 1     // R0<-W
10: MOVF     R1 0     // W<-R1
11: XORWF    R4 0     // W<-R4^W
12: ANDWF    R11 0    // W<-R11&W
13: IORWF    R0 0     // W<-R0|W
14: MOVWF    R11 1    // R11<-W
15: MOVF     R2 0     // W<-R2
16: XORWF    R5 0     // W<-R5^W
17: XORWF    R11 0    // W<-R11^W
18: MOVWF    R9 1     // R9<-W
19: MOVF     R2 0     // W<-R2
20: ANDWF    R5 0     // W<-R5&W
21: MOVWF    R0 1     // R0<-W
22: MOVF     R2 0     // W<-R2
23: XORWF    R5 0     // W<-R5^W
24: ANDWF    R11 0    // W<-R11&W
25: IORWF    R0 0     // W<-R0|W
26: MOVWF    R11 1    // R11<-W
27: MOVF     R3 0     // W<-R3
28: XORWF    R6 0     // W<-R6^W
29: XORWF    R11 0    // W<-R11^W
30: MOVWF    R10 1    // R10<-W
31: MOVF     R3 0     // W<-R3
32: ANDWF    R6 0     // W<-R6&W
33: MOVWF    R0 1     // R0<-W
34: MOVF     R3 0     // W<-R3
35: XORWF    R6 0     // W<-R6^W
36: ANDWF    R11 0    // W<-R11&W
37: IORWF    R0 0     // W<-R0|W
38: MOVWF    R11 1    // R11<-W
```

図 B.6　B2 (7bit-DigitalAdder) の初期プログラム

```
 1: CLRF     R0 1// R0<-0
 2: MOVLW    1   // W<-1
 3: MOVWF    R7 1// R7<-W (R7<-1)
 4: MOVLW    1   // W<-1
 5: XORWF    R1 0// W<-R1^W (W<-not R1)
 6: ANDWF    R7 1// R7<-R7&W
 7: MOVLW    1   // W<-1
 8: XORWF    R2 0// W<-R2^W (W<-not R2)
 9: ANDWF    R7 1// R7<-R7&W
10: MOVF     R3 0// W<-R3
11: ANDWF    R7 0// W<-R7&W
12: IORWF    R0 1// R0<-R0|W
13: MOVLW    1   // W<-1
14: MOVWF    R7 1// R7<-W (R7<-1)
15: MOVLW    1   // W<-1
16: XORWF    R1 0// W<-R1^W (W<-not R1)
17: ANDWF    R7 1// R7<-R7&W
18: MOVF     R2 0// W<-R2
19: ANDWF    R7 1// R7<-R7&W
20: MOVF     R4 0// W<-R4
21: ANDWF    R7 0// W<-R7&W
22: IORWF    R0 1// R0<-R0|W
23: MOVLW    1   // W<-1
24: MOVWF    R7 1// R7<-W (R7<-1)
25: MOVF     R1 0// W<-R1
26: ANDWF    R7 1// R7<-R7&W
27: MOVLW    1   // W<-1
28: XORWF    R2 0// W<-R2^W (W<-not R3)
29: ANDWF    R7 1// R7<-R7&W
30: MOVF     R5 0// W<-R5
31: ANDWF    R7 0// W<-R7&W
32: IORWF    R0 1// R0<-R0|W
33: MOVLW    1   // W<-1
34: MOVWF    R7 1// R7<-W (R7<-1)
35: MOVF     R1 0// W<-R1
36: ANDWF    R7 1// R7<-R7&W
37: MOVF     R2 0// W<-R2
38: ANDWF    R7 1// R7<-R7&W
39: MOVF     R6 0// W<-R6
40: ANDWF    R7 0// W<-R7&W
41: IORWF    R0 1// R0<-R0|W
```

図 B.7　B3 (6bit-Multiplexer) の初期プログラム

```
 1: CLRF    R0 1 // R0<-0
 2: MOVF    R1 0 // W<-R1
 3: ANDWF   R2 0 // W<-R2&W
 4: ANDWF   R3 0 // W<-R3&W
 5: ANDWF   R4 0 // W<-R4&W
 6: IORWF   R0 1 // R0<-R0|W
 7: MOVF    R1 0 // W<-R1
 8: ANDWF   R2 0 // W<-R2&W
 9: ANDWF   R3 0 // W<-R3&W
10: ANDWF   R5 0 // W<-R5&W
11: IORWF   R0 1 // R0<-R0|W
12: MOVF    R1 0 // W<-R1
13: ANDWF   R2 0 // W<-R2&W
14: ANDWF   R3 0 // W<-R3&W
15: ANDWF   R6 0 // W<-R6&W
16: IORWF   R0 1 // R0<-R0|W
17: MOVF    R1 0 // W<-R1
18: ANDWF   R2 0 // W<-R2&W
19: ANDWF   R3 0 // W<-R3&W
20: ANDWF   R7 0 // W<-R7&W
21: IORWF   R0 1 // R0<-R0|W
22: MOVF    R1 0 // W<-R1
23: ANDWF   R2 0 // W<-R2&W
24: ANDWF   R4 0 // W<-R4&W
25: ANDWF   R5 0 // W<-R5&W
26: IORWF   R0 1 // R0<-R0|W
27: MOVF    R1 0 // W<-R1
28: ANDWF   R2 0 // W<-R2&W
29: ANDWF   R4 0 // W<-R4&W
30: ANDWF   R6 0 // W<-R6&W
31: IORWF   R0 1 // R0<-R0|W
32: MOVF    R1 0 // W<-R1
33: ANDWF   R2 0 // W<-R2&W
34: ANDWF   R4 0 // W<-R4&W
35: ANDWF   R7 0 // W<-R7&W
36: IORWF   R0 1 // R0<-R0|W
37: MOVF    R1 0 // W<-R1
38: ANDWF   R2 0 // W<-R2&W
39: ANDWF   R5 0 // W<-R5&W
40: ANDWF   R6 0 // W<-R6&W
41: IORWF   R0 1 // R0<-R0|W
42: MOVF    R1 0 // W<-R1
43: ANDWF   R2 0 // W<-R2&W
44: ANDWF   R5 0 // W<-R5&W
45: ANDWF   R7 0 // W<-R7&W
46: IORWF   R0 1 // R0<-R0|W
47: MOVF    R1 0 // W<-R1
48: ANDWF   R2 0 // W<-R2&W
49: ANDWF   R6 0 // W<-R6&W
50: ANDWF   R7 0 // W<-R7&W
51: IORWF   R0 1 // R0<-R0|W
52: MOVF    R1 0 // W<-R1
53: ANDWF   R3 0 // W<-R3&W
54: ANDWF   R4 0 // W<-R4&W
55: ANDWF   R5 0 // W<-R5&W
56: IORWF   R0 1 // R0<-R0|W
57: MOVF    R1 0 // W<-R1
58: ANDWF   R3 0 // W<-R3&W
59: ANDWF   R4 0 // W<-R4&W
60: ANDWF   R6 0 // W<-R6&W
61: IORWF   R0 1 // R0<-R0|W
62: MOVF    R1 0 // W<-R1
63: ANDWF   R3 0 // W<-R3&W
64: ANDWF   R4 0 // W<-R4&W
65: ANDWF   R7 0 // W<-R7&W
66: IORWF   R0 1 // R0<-R0|W
67: MOVF    R1 0 // W<-R1
68: ANDWF   R3 0 // W<-R3&W
69: ANDWF   R5 0 // W<-R5&W
70: ANDWF   R6 0 // W<-R6&W
```

図 B.8　B4 (7bit-Majority) の初期プログラム（1/3）

```
 71: IORWF   R0 1 // R0<-R0|W
 72: MOVF    R1 0 // W<-R1
 73: ANDWF   R3 0 // W<-R3&W
 74: ANDWF   R5 0 // W<-R5&W
 75: ANDWF   R7 0 // W<-R7&W
 76: IORWF   R0 1 // R0<-R0|W
 77: MOVF    R1 0 // W<-R1
 78: ANDWF   R3 0 // W<-R3&W
 79: ANDWF   R6 0 // W<-R6&W
 80: ANDWF   R7 0 // W<-R7&W
 81: IORWF   R0 1 // R0<-R0|W
 82: MOVF    R1 0 // W<-R1
 83: ANDWF   R4 0 // W<-R4&W
 84: ANDWF   R5 0 // W<-R5&W
 85: ANDWF   R6 0 // W<-R6&W
 86: IORWF   R0 1 // R0<-R0|W
 87: MOVF    R1 0 // W<-R1
 88: ANDWF   R4 0 // W<-R4&W
 89: ANDWF   R5 0 // W<-R5&W
 90: ANDWF   R7 0 // W<-R7&W
 91: IORWF   R0 1 // R0<-R0|W
 92: MOVF    R1 0 // W<-R1
 93: ANDWF   R4 0 // W<-R4&W
 94: ANDWF   R6 0 // W<-R6&W
 95: ANDWF   R7 0 // W<-R7&W
 96: IORWF   R0 1 // R0<-R0|W
 97: MOVF    R1 0 // W<-R1
 98: ANDWF   R5 0 // W<-R5&W
 99: ANDWF   R6 0 // W<-R6&W
100: ANDWF   R7 0 // W<-R7&W
101: IORWF   R0 1 // R0<-R0|W
102: MOVF    R2 0 // W<-R2
103: ANDWF   R3 0 // W<-R3&W
104: ANDWF   R4 0 // W<-R4&W
105: ANDWF   R5 0 // W<-R5&W
106: IORWF   R0 1 // R0<-R0|W
107: MOVF    R2 0 // W<-R2
108: ANDWF   R3 0 // W<-R3&W
109: ANDWF   R4 0 // W<-R4&W
110: ANDWF   R6 0 // W<-R6&W
111: IORWF   R0 1 // R0<-R0|W
112: MOVF    R2 0 // W<-R2
113: ANDWF   R3 0 // W<-R3&W
114: ANDWF   R4 0 // W<-R4&W
115: ANDWF   R7 0 // W<-R7&W
116: IORWF   R0 1 // R0<-R0|W
117: MOVF    R2 0 // W<-R2
118: ANDWF   R3 0 // W<-R3&W
119: ANDWF   R5 0 // W<-R5&W
120: ANDWF   R6 0 // W<-R6&W
121: IORWF   R0 1 // R0<-R0|W
122: MOVF    R2 0 // W<-R2
123: ANDWF   R3 0 // W<-R3&W
124: ANDWF   R5 0 // W<-R5&W
125: ANDWF   R7 0 // W<-R7&W
126: IORWF   R0 1 // R0<-R0|W
127: MOVF    R2 0 // W<-R2
128: ANDWF   R3 0 // W<-R3&W
129: ANDWF   R6 0 // W<-R6&W
130: ANDWF   R7 0 // W<-R7&W
131: IORWF   R0 1 // R0<-R0|W
132: MOVF    R2 0 // W<-R2
133: ANDWF   R4 0 // W<-R4&W
134: ANDWF   R5 0 // W<-R5&W
135: ANDWF   R6 0 // W<-R6&W
136: IORWF   R0 1 // R0<-R0|W
137: MOVF    R2 0 // W<-R2
138: ANDWF   R4 0 // W<-R4&W
139: ANDWF   R5 0 // W<-R5&W
140: ANDWF   R7 0 // W<-R7&W
```

図 B.8　B4 (7bit-Majority) の初期プログラム（2/3）

```
141: IORWF   R0 1 // R0<-R0|W
142: MOVF    R2 0 // W<-R2
143: ANDWF   R4 0 // W<-R4&W
144: ANDWF   R6 0 // W<-R6&W
145: ANDWF   R7 0 // W<-R7&W
146: IORWF   R0 1 // R0<-R0|W
147: MOVF    R2 0 // W<-R2
148: ANDWF   R5 0 // W<-R5&W
149: ANDWF   R6 0 // W<-R6&W
150: ANDWF   R7 0 // W<-R7&W
151: IORWF   R0 1 // R0<-R0|W
152: MOVF    R3 0 // W<-R3
153: ANDWF   R4 0 // W<-R4&W
154: ANDWF   R5 0 // W<-R5&W
155: ANDWF   R6 0 // W<-R6&W
156: IORWF   R0 1 // R0<-R0|W
157: MOVF    R3 0 // W<-R3
158: ANDWF   R4 0 // W<-R4&W
159: ANDWF   R5 0 // W<-R5&W
160: ANDWF   R7 0 // W<-R7&W
161: IORWF   R0 1 // R0<-R0|W
162: MOVF    R3 0 // W<-R3
163: ANDWF   R4 0 // W<-R4&W
164: ANDWF   R6 0 // W<-R6&W
165: ANDWF   R7 0 // W<-R7&W
166: IORWF   R0 1 // R0<-R0|W
167: MOVF    R3 0 // W<-R3
168: ANDWF   R5 0 // W<-R5&W
169: ANDWF   R6 0 // W<-R6&W
170: ANDWF   R7 0 // W<-R7&W
171: IORWF   R0 1 // R0<-R0|W
172: MOVF    R4 0 // W<-R4
173: ANDWF   R5 0 // W<-R5&W
174: ANDWF   R6 0 // W<-R6&W
175: ANDWF   R7 0 // W<-R7&W
176: IORWF   R0 1 // R0<-R0|W
```

図 B.8　B4 (7bit-Majority) の初期プログラム（3/3）