特別研究報告

題目

メタヒューリスティクスを用いた集約可能コードクローン量の推定

指導教員
井上 克郎 教授

報告者
石津 卓也

平成 28 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 27 年度 特別研究報告

メタヒューリスティクスを用いた集約可能コードクローン量の推定

石津 卓也

内容梗概

ソフトウェアの保守はソースコードの行数に依存して困難なものになる．行数を増大させる要因の１つとしてコードクローンが挙げられる．コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことを示しており, 既存コードのコピーアンドペーストによる再利用や, 特定のコードを意図的に繰り返し書くことなどによって生じる．

コードクローンに対する保守作業の１つとして集約が挙げられる. 集約とは, 互いにコードクローンとなっているコード片を１つのメソッドやクラスなどにまとめることである. また，集約可能なコードクローンを集約して減る行数を集約可能コードクローン量と呼ぶことにする．集約を行うことで, 保守の対象となるコードクローンを除去することが可能となる. そのため，コードクローンを集約することは，保守するソースコードの行数の減少につながるので，ソースコードの保守が容易になる．

ただし，開発者がコードクローンの集約をするにあたって，コードクローンをどこに１つに集約させるか，あるいは，変数名や変数の型などに差異があるコードクローンをどう集約するかを開発者は逐一ソースコードを読んで判断しなくてはならない．開発者が集約を行うのに必要なコストを鑑みたときに，コードクローンの集約がソフトウェアの保守の労力の削減に有効な作業であるのか判断が難しい．したがって，集約により減少する行数を推定することは，開発者が集約を行うか判断する材料になりうるので，集約を行う労力を削減するのに有効であると考えられる．

コードクローンのコード片単位で部分的に重複する状態をコードクローンがオーバーラップしているという．複数のコードクローン同士がオーバーラップしている場合，１つのコードクローンを集約すると，オーバーラップしている他のコードクローンは集約できなくなる．したがって，どどのコードクローンを集約するかによって，最終的な集約可能コードクローン量は変化する．そのため，最大の集約可能コードクローン量を推定するには，コードクローンを集約することにおいて，全通りの集約可能コードクローン量を推定する必要がある．しかし，全通りの集約可能コードクローン量に対して推定するのは現実的ではないので，計算方法を工夫する必要がある．

本研究では，適用実験としてオーバーラップしているコードクローンを含むソフトウェアを対象に，アルゴリズムを用いて集約可能なコードクローン量を推定して，一般的にメタ

ヒューリスティクスで用いられる 4 つのアルゴリズムが示した集約可能コードクローン量を比較した．その結果，遺伝的アルゴリズムが集約可能コードクローン量の推定に適用するのが最も高い集約可能コードクローン量を示すことでがわかった．また，オーバーラップの制約が原因で集約不可だったクローンセットを集約可能になるようにクローンセットを分割する手法を行い，クローンセット分割の有用性を示した．

主な用語

コードクローン
メタヒューリスティクス
コードクローンのオーバーラップ

目 次

1 まえがき 5

2 背景 7
- 2.1 コードクローン 7
  - 2.1.1 コードクローンの検出ツール 7
  - 2.1.2 コードクローンの集約 8
  - 2.1.3 コードクローンのオーバーラップ 9
  - 2.1.4 集約可能コードクローン量推定の必要性 9
- 2.2 メタヒューリスティクス 11
  - 2.2.1 メタヒューリスティクスで用いられるアルゴリズム 11
  - 2.2.2 問題の記述法 14
- 2.3 探索的手法に基づくソフトウェア工学に関する関連研究 15

3 提案手法 17
- 3.1 オーバーラップの抽出 18
- 3.2 集約可能コードクローン量の推定 19
  - 3.2.1 オーバーラップを含む集約可能コードクローン量の推定 20
  - 3.2.2 クローンセットの分割 21
- 3.3 探索的手法に基づくソフトウェア工学への適用 23
  - 3.3.1 Representation 23
  - 3.3.2 Operators 24
  - 3.3.3 Fitness Function 25

4 適用実験 27
- 4.1 対象データ 27
- 4.2 各アルゴリズムによる集約可能コードクローン量推定結果の比較 27
- 4.3 クローンセットの分割に要する実行時間 35
- 4.4 実行時間あたりの集約可能コードクローン量の比較 35
- 4.5 考察 40

5 まとめと今後の課題 43

謝辞 44

参考文献 **45**

1　まえがき

ソフトウェアの保守はソースコードの行数に依存して困難なものになる．行数を増大させる要因の１つとしてコードクローンが挙げられる．コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことを示している [7] [9]．コードクローンは既存コードのコピーアンドペーストによる再利用や, 特定のコードを意図的に繰り返し書くことなどによって生じる [9].

コードクローンに対する保守作業として集約が挙げられる．集約とは，互いにコードクローンとなっているコード片の集合（以下，クローンセットと呼ぶ）に対して，クローンセットに含まれるコードクローンを１つのクラスやメソッドにまとめることである [1] [7].また，集約可能なコードクローンを集約して減る行数を集約可能コードクローン量と呼ぶことにする．集約を行うことで，保守の対象であるコードクローンを除去できる．そのため，コードクローンを集約することで，保守するソースコードの行数が減少して，ソースコードの保守が容易になる．

コードクローンの集約を行うには，コードクローンをどこに集約するか，あるいは，変数名や変数の型，予約語などに差異があるコードクローンをどう集約するか，開発者が逐一ソースコードを読んで，コードクローンが集約の対象になりうるのか判断をしなければならない．コードクローンが多く，開発者が判断しなければならない機会が多いソースコードは,集約により減らせるプログラム全体の集約可能コードクローン量がわかりにくかったり，その判断にかかる費用や期間が見えにくかったりするので，集約の計画が立たなくて開発を敬遠しがちである．そこで，コードクローンに対する集約による効果を事前に推定して集約を支援することを考える．開発者は事前に集約可能コードクローン量を知ることで率先して集約を実行する．

集約可能コードクローン量を推定する場合，コードクローンのコード片がオーバーラップしていることがあるために推定が困難になる場合がある．コードクローンのオーバーラップとは，複数のコードクローンのコード片が部分的に重複していることを指す．コードクローンがオーバーラップしている場合，一方のコードクローンを集約するとオーバーラップをしていたコードクローンはコード片が部分的に欠損するため，集約が不可能になる．そのため,コードクローンのオーバーラップは集約可能コードクローン量の推定を困難にする．

コードクローンがオーバーラップしている場合，集約するコードクローンの組み合わせによって集約可能コードクローン量は変化する．よって，集約するコードクローンの組み合わせに対して，全通りを計算して最善だった順番を採用することが考えられる．しかし，実際のソースコードにおいて，コードクローンを選択する組み合わせは時間的に計算するのが困難なほど現実的ではない．そこで，効率的にどのコードクローンを集約するのかを求める探

索手法が必要である．

本研究では，適用実験としてオーバーラップを考慮したコードクローンを含むソフトウェアを対象にアルゴリズムを用いて集約可能なコードクローン量を計算して，一般的にメタヒューリスティクスで用いられる 4 つのアルゴリズムを比較している．メタヒューリスティクスであるアルゴリズムには貪欲法や山登り法，焼きなまし法，遺伝的アルゴリズムが代表として挙げられる [14]．例えば，集約可能コードクローン量の大きなクローンセットから集約を行う貪欲法では，実行時間は短いがオーバーラップを含んだ集約では必ずしも無駄の少ない選択をできるとは限らない．コードクローンに対する集約を行う各アルゴリズムの実行時間や集約可能コードクローン量の比較を行う．また，結果として遺伝的アルゴリズムがオーバーラップを考慮した集約可能コードクローン量を求めるうえで最も高い集約可能コードクローン量を示すアルゴリズムと判明した．

また，各アルゴリズムはコードクローンに対する集約をクローンセット単位で行う．その際，オーバーラップしているクローンセットは集約されないが，クローンセット内に集約できるコードクローンが存在している場合がある．この問題を解決するために，クローンセットを，オーバーラップしているコードクローン毎に分割することを考える．また，クローンセット分割を行う場合とおこなわない場合の適用実験で比較してクローンセット分割の有用性を示した．

以降，2 節では本研究の背景について説明する．3 節では本研究で用いた手法，4 節では，本研究で用いた手法に関する適用実験の結果をまとめて，5 節で，そのまとめと今後の課題を示す．

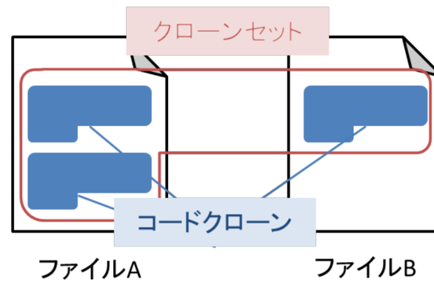

図 1: コードクローンの例

2　背景

本章では，本研究の背景としてコードクローンとその検出ツール，オーバーラップ，クローンセットの分割について説明する．

2.1　コードクローン

コードクローンとは，プログラムテキスト中の同一，あるいは，類似したコードの断片を意味する [9]．図 1 は，ファイル A とファイル B に存在する同一コード片がコードクローンであることを示している．また，一般的に，同一コードクローンから構成される集合をクローンセット (Cloneset) と呼ぶ．コードクローンの存在はソフトウェアの保守作業を困難なものにすると言われている．例えば，あるコード片を修正する場合，コードクローンに関しても同様の修正が行われる可能性がある．コードクローンに気づかずにソースコードの修正を疎かにすると，ソースコードにバグが混入する可能性がある．そのため，コードクローンの存在を知ることはソフトウェアの保守の観点からすると重要なことである．しかし，ソフトウェアの規模が大きい場合，開発者がソースコードを読んですべてのコードクローンの存在を知ることは非効率的であるため，一般的にはコードクローンを自動で検出するツールが用いられる．

2.1.1　コードクローンの検出ツール

コードクローンの検出手法として，プログラムの字句解析による行単位の検出や字句単位の検出，特徴メトリクスを用いた検出などがある [9] [10]．行単位の検出ではハッシュ関数を用いて，プログラムテキストの各行をハッシュ値に変換して，そのハッシュ値の列を対象

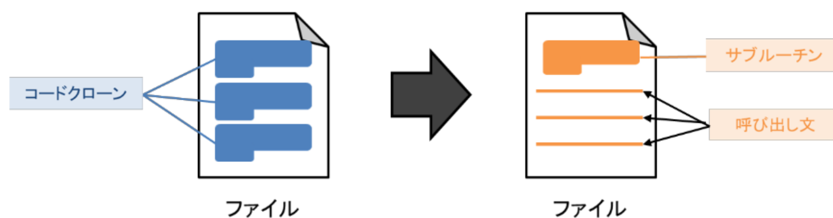

図 2: コードクローンの集約例

として類似したハッシュ値列を求めることにより，コードクローンを発見する手法である．字句単位の検出では，閾値以上連続して一致する字句の部分列がコードクローンとして検出される．行単位で検出するよりも検出粒度が細かく，コーディングスタイルに依存しないなどの特徴を持つ．特徴メトリクスを用いたコードクローン検出では，ファイルやクラス，メソッドなどのモジュールに対してメトリクスを計測し，その値の一致または近似の度合を調べるすることで，モジュール単位でのコードクローン検出を可能にする．

字句解析ベースのコードクローン検出ツールとして，CCFinder がある [11]．CCFinder は高いスケーラビリティを有しており, 大規模なソフトウェアに対しても実用的な時間でコードクローン検出が可能である． また, 変数名や関数名といったユーザ定義名や，変数の型などの一部予約語の違いなどの表現上の差異があるコードクローンを検出することができるという特徴も備えている [12] [16]．CCFinder は様々な大規模ソフトウェアに適用され, その有用性が確認されている [6]．

2.1.2 コードクローンの集約

コードクローンに対する保守作業として集約が挙げられる．集約とは，クローンセットに対して，クローンセットに含まれるコードクローンを 1 つのクラスやメソッドなどのサブルーチンにまとめて，コードクローンを呼び出し文に置き換えることである [7]．図 2 は，コードクローンが 1 つのサブルーチンに集約されて，コードクローンを呼び出し文に置き換えている様子を表している．集約を行うことで保守の対象であるコードクローンを除去できる．そして，コードクローンの除去により，ソースコードの同時修正を行う手間が省ける．そのため，コードクローンの集約はソースコードの保守作業の労力を軽減する役割を持つ．

コードクローンをどう集約するかについては，様々な方法が提案されている [8]．例えば，メソッドを抽出する方法や，メソッドを親クラスに引き上げる方法などが挙げられる．メ

ソッドの抽出とは，既存のメソッドの一部分であるコードクローンのコード片に対して，新たなメソッドを１つ用意して，すべてのコードクローンの共通メソッドとして代替することである．新たに要したメソッドには，コードクローン間の変数やリテラルといった差異部分を引数として引き渡す．また，メソッドの引き上げとは，共通の親クラスを持つ，複数の子クラスに重複したメソッドが存在する場合，それらの共通した親クラスに引き上げることにより，コードクローンを除去する．

2.1.3 コードクローンのオーバーラップ

コードクローンのオーバーラップとは，複数のコードクローンのコード片が部分的に重複していることを指す．コードクローンがオーバーラップしている場合，一方のコードクローンを集約するとオーバーラップしていたコードクローンはコード片が部分的に欠損するため，集約が不可能になる．

図 3 は例を示している．2 つのコードクローン a1，b1 がオーバーラップしている例を示している．青で塗られたコード片 a1 と赤で塗られたコード片 b1 はそれぞれ異なるクローンセットに属しており，2 つのコードクローンが部分的に重複している様子がわかる．

2.1.4 集約可能コードクローン量推定の必要性

コードクローンの集約を行うには，コードクローンをどこに集約するか，あるいは，変数名や変数の型，予約語といったコードクローンの差異をどう吸収するか，開発者が逐一ソースコードを読んで，コードクローンが集約の対象になりうるのか判断をしなければならない．コードクローンが多く，開発者が判断しなければならない機会が多いソースコードは，集約により減らせるプログラム全体の集約可能コードクローン量がわかりにくかったり，その判断にかかる費用や期間が予想しずらかったりするので，集約の計画が立たなくて開発を敬遠しがちである．そこで，コードクローンに対する集約による効果を事前に推定して集約を支援することを考える．開発者は事前に集約可能コードクローン量を知ることで率先して集約を実行する．

オーバーラップを考慮したコードクローンの集約可能コードクローン量を推定するには，集約するコードクローンの順番によって集約可能コードクローン量が変化することを考慮する必要がある．理想は集約するコードクローンの組み合わせに対して，全通りを推定して最善だった順番を採用すればいい．しかし，実際のソースコードにおいて，コードクローンを選択する組み合わせは時間的に計算するのが困難なほど現実的ではない．そこで，集約するコードクローンを選ぶ順番を求める問題に対して，既存の探索的手法の適用が考えられる．

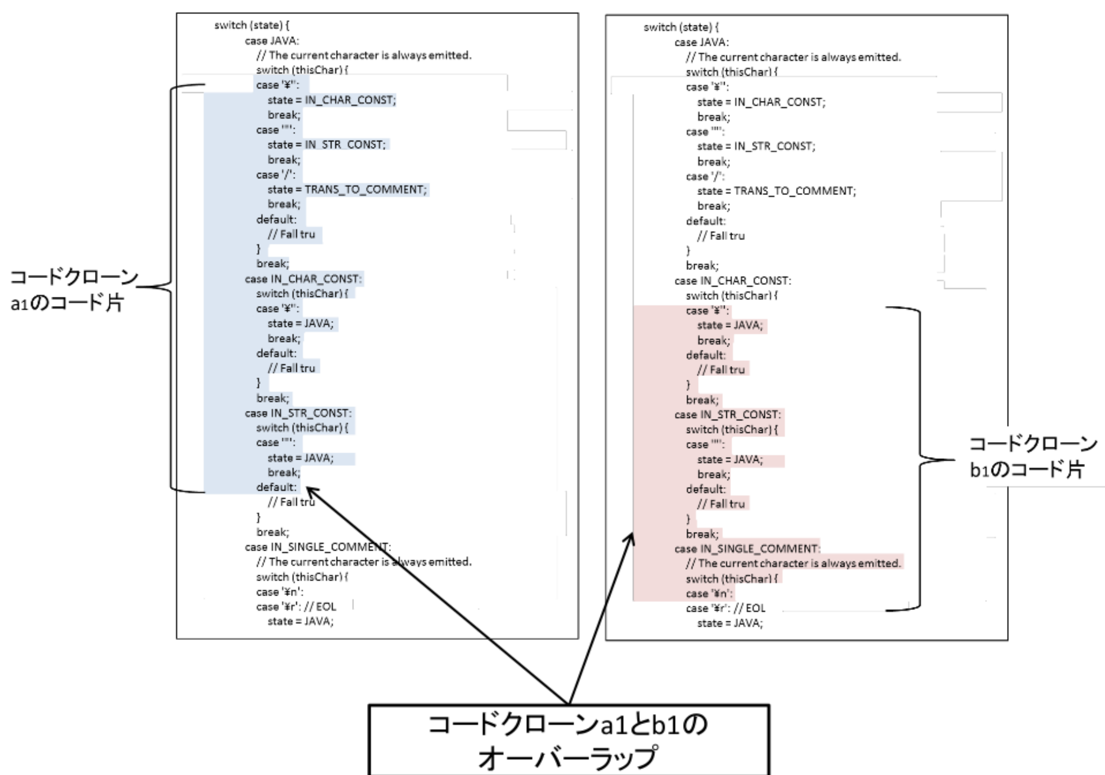

図 3: 同一ファイル内のコード片で発生した，コードクローン a1 とコードクローン b1 のオーバーラップの例

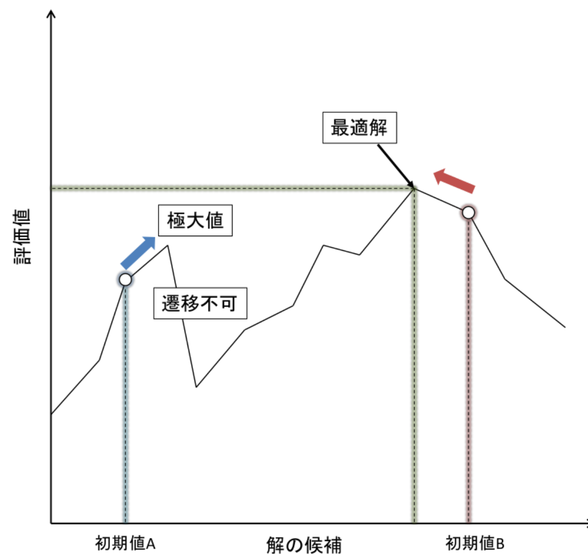

図 4: 山登り法 (HC) の例

2.2 メタヒューリスティクス

メタヒューリスティクスとは，特定の問題に依存しない，組み合わせ最適化問題における，近似解を求める解法をもつアルゴリズムの基本的な枠組みのことである．アルゴリズムとして，局所的な探索に基づくアルゴリズムや，大域的な，個体群に基づくアルゴリズムが挙げられる．局所的な探索に基づくアルゴリズムとは，現在の解から近傍を求めて，現在の解を近傍の解に更新する作業を繰り返すことで近似解を求めるアルゴリズムである．大域的な個体群に基づくアルゴリズムとは，解の候補を生物の個体群に見立て最適解を求めるアルゴリズムである．このアルゴリズムでは，個体が次の世代に生き残るのかを決定する関数を工夫して生物の生殖や自然淘汰，突然変異などを再現する．これを繰り返して残った個体が最適解に近似する．

2.2.1 メタヒューリスティクスで用いられるアルゴリズム

本研究では，コードクローンの集約可能コードクローン量をメタヒューリスティクスで一般的に用いられる 4 つのアルゴリズムに適用して，それらの評価を比較する．メタヒューリスティクスで用いる 4 つのアルゴリズムとして貪欲法 (Greedy Algorithm)，山登り法 (Hill Climbing)，焼きなまし法 (Simulated Annealing)，遺伝的アルゴリズム (Genetic Algorithm) が代表的なアルゴリズムとして挙げられる．HC や SA は局所的な探索アルゴリズムであり，GA は大域的な，個体群に基づく探索アルゴリズムである．

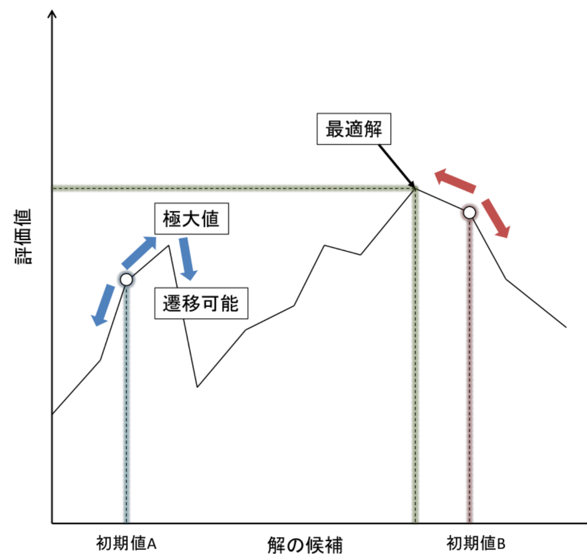

図 5: 焼きなまし法 (SA) の例

貪欲法

貪欲法 (Greedy Algorithm) は，最適化問題に対して，解の候補からその解の評価値を求めて，評価値の高い順番に解の候補を選択することで最適解を求めるアルゴリズムである [2]．最適化問題によっては最適解を得ることは難しいが，簡単に近似解を得ることができるアルゴリズムなので，最適化問題に対する近似解を求めるために適用されることが多い．

山登り法

HC は現在の解の候補の評価値と，近傍の解の候補の評価値を比較して，最適解に近い評価値を示せば現在の最適解に更新する [5]．そして，近傍の中に現在の解の候補の評価値を上回る解がなければ解の更新はされずに，現在の解を最適解とするアルゴリズムである．そのため，HC では初期値の決め方が重要で，初期値次第では最適解にたどり着けない可能性がある．そこで，評価関数を求める評価関数を上に凸であるように決めることで，局所解で動けなくなる事態を排除できる．

図 4 は，HC の局所的な探索をしている様子を表している．解の候補を評価関数で評価した値を，縦軸で評価値として表している．図 4 で示しているように，与えられた問題は最適解をもっている．しかし，初期値 A では，極大値に陥り最適解が得られない様子がわかる．一方で初期値 B では，最適解を得る．そのため，初期値の決め方に依存するアルゴリズムといえる．

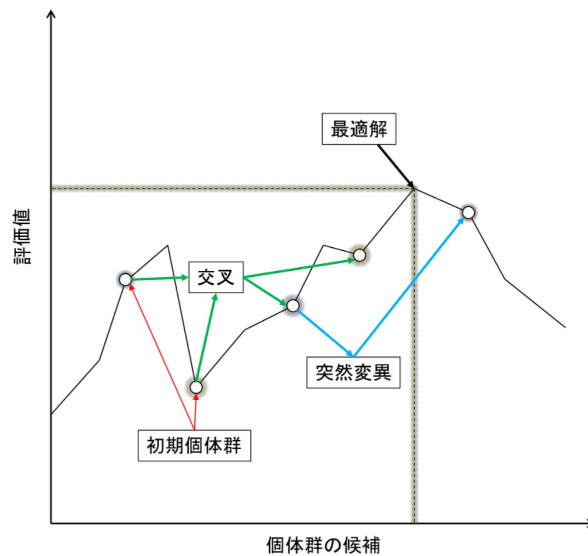

図 6: 遺伝的アルゴリズム (GA) の例

焼きなまし法

SA は現在の解の候補から近傍の解をランダムに 1 つだけ決定して，その近傍の解の候補の評価値と現在の解の候補の評価値を比較する [15]．近傍の方が良ければ、近傍に移動するが，そうでなくてもランダムで近傍に移動するアルゴリズムである．また，焼きなまし法では，温度を表す変数を用いる．評価開始時点では高い温度が設定され，評価が進むごとに温度が下がっていく．温度が高いほど，評価が下がっても新しい解を採用する確率が上がり，温度が下がると採用する確率が下がる．また，SA には明確な修了基準がないので，終了条件として評価回数を設定できる．HC と異なる点は，ランダムで解の候補が近傍に移動するため，極大値で動けなくなることがないことである．

図 5 は，SA の局所的な探索をしている様子を表したものである．初期値 A でも，極大値を経た後でも遷移することで，最適解を得る可能性があることがわかる．同時に初期値 B のような最適解の近傍であっても，最適解にはたどり着けない可能性もある．

遺伝的アルゴリズム

GA は先に説明した通り，解の候補を生物の個体に見立てることで，世代交代を繰り返して生き残った個体群から最適解を求める手法である [3]．また，GA は無限に評価できるので，終了条件として評価する回数の制限が設けられている．GA は，解の候補を発見する手法が大域的であるので，近傍の評価値が大きく増減するような評価関

数を持つ問題に対して有効なアルゴリズムである．

図6は，GAの大域的な探索をしている様子を表したものである．初期個体群が与えられると，世代交代の際に個体群同士での交叉が起こり，新たな個体群が得られる．また，交叉だけではなく突然変異によっても新たな個体が得られる．このような操作を繰り返して，最適解を得られる．

本研究では，遺伝的アルゴリズムとして，多目的最適化問題によく用いられるNSGAII（Non-dominated Sorting Genetic Algorithms-II）アルゴリズムを使用した．多目的最適化問題とは，目的関数を2つ以上持っている最適化問題のことである．本研究では，目的関数が集約可能コードクローン量の1つしかない単一目的最適化問題を扱うが，NSGAIIは目的関数が1つでも適用可能である．また，NSGAIIアルゴリズムの使用にあたって，JavaのフレームワークとしてMOEAを利用した [13]．MOEAでは，多目的最適化問題に対応する，遺伝的アルゴズムや遺伝的プログラミングがオープンソフトウェアとして組み込まれている．

2.2.2 問題の記述法

Search Based Software Engineering (SBSE) とは，ソフトウェア工学上で生じるさまざな問題に対して，メタヒューリスティクスの探索的手法を適用する技術のことである．ソフトウェア工学で生じる問題を解決する手法としては，従来的には線形計画や動的計画などの数学的，あるいは統計的なオペレーションズリサーチが挙げられる．しかし，オペレーションズリサーチは，ソフトウェア工学で生じる最適化問題に対して，その計算が大規模になるためにあまり実用的でないことが知られている [4].

メタヒューリスティクスは，特定の問題に依存しない手法のみに焦点を当てた考え方であり，あらゆる問題に対して汎用的に対応することが求められる．そのためメタヒューリスティクスでは，与えられた問題ごとにRepresentation，Operators，Fitness Functionを定義することがSBSEでは推奨される．これらは以下のように定義される．

- Representationとは，与えられた問題における，遷移可能な状態のことである．
- Operatorsとは，各状態に対して実行可能な操作のことである．
- Fitness Functionとは，現状の解が与えられた問題の最適解にどれほど近づいているのかを評価する関数のことである．

例えば，1から9までの数字がランダムに一列に並んでいたときに，それらの数字に対して隣り合っている数字を入れ替える操作だけが許されている条件で，数字を昇順にソートする

問題を考える．この問題をメタヒューリスティックを用いて解を求めるとする．このとき，この問題は，Representation，Operators，Fitness Function を用いて，次のように定義される．

Representation

はじめに，Representation を R(k)(k は 0 以上の整数) とし，一列に並んでいる数字がどの順番で並んでいるのかを表現する．k 回目の数字の入れ替えがあった後の，一列に並ぶ数字の状態を R(k) とする．

$$\text{定義の例：} R(k) = \{2, 5, 6, 1, 8, 3, 9, 7, 4\}$$

Operators

隣り合っている数字を入れ替える操作だけが許されているので，次のように表現できる．R の状態において，左から i 番目と i+1 番目の数字を入れ替える操作を Opr(R,i) で表す．例では i=5 より左から 5 番目の数字と 6 番目の数字を入れ替える操作になる．

$$\text{定義の例：} \quad R(k+1) = Opr(R(k+1), 5) = \{2, 5, 6, 1, 3, 8, 9, 7, 4\}$$

Fitness Function

例のソート問題における，最終的な解を考えると，左から i 番目の数字は i である．よって，i 番目の数字を現状の解からの差の絶対値の和で評価する事が考えられる．すなわち，1 番目から 9 番目の数字の評価の総和が 0 であるとき，ソートが終了したことになる．例における，k 回目の評価値より k+1 回目の評価値の方が最適解 0 に近い評価値をとっているので，k+1 回目の方が最適解に近いと判断できる．ただし，図 4，図 5，図 6 で示したように，必ずしも評価値に近づくことが最適解を得られる保証はない．

$$\text{定義の例：} F(R(m)) = \sum_{m=1}^{9} |m - a[m]| \ \ (a[m] \text{ は } m \text{ 番目の要素})$$

$$F(R(k)) = |1-2| + |2-5| + |3-6| + |4-1| + |5-8| + |6-3| + \cdots + |9-4| = 24,$$

$$F(R(k+1)) = |1-2| + |2-5| + |3-6| + |4-1| + |5-3| + |6-8| + \cdots + |9-4| = 23$$

2.3 探索的手法に基づくソフトウェア工学に関する関連研究

Bauktif らは，単一プラットフォームとして開発されたソフトウェアが新しいプロセッサに移植した際に生じるソフトウェアの品質の低下を改善するために，複数の制約を持つナップサック問題やコードクローンリファクタリングのモデル化とそれに必要な労力を推定する研究を行った [1]．最初のバージョンが 1992 年にリリースされた GRASS ソフトウェアを対

象に，ソースコードの管理による，関数単位で生じるコードクローンのリファクタリングのモデル化を GA を基に実践している．さらに，Bauktif らは，ソフトウェアの品質向上に関する一般的な問題は，メタヒューリスティクスを用いた優先順位や制約の下でスケジュールするべきと述べている．

O'Keeffe らは，機能追加を繰り返すオブジェクト指向型システムが，一般的に設計品質が低下する問題を解決するためにリファクタリングが有効であると考え，検索に基づいたリファクタリングにおいて，SA，GA，HC の実証的比較の結果を示した [14]．Mark 氏らは，入力プログラムとして商用プログラムより設計品質の低下が少なく，また商用プログラムに比べて規模が小さい 5 つのオープンソースを用いて実験を行った．また，GA 探索には CODe-Imp と呼ばれるリファクタリングツールを拡張したツールを利用している．O'Keeffe らは，HC が入力プログラムに特定の特性を有していたことに触れつつも，HCM が検索ベースのリファクタリングの最適な検索技術であると結論付けている．

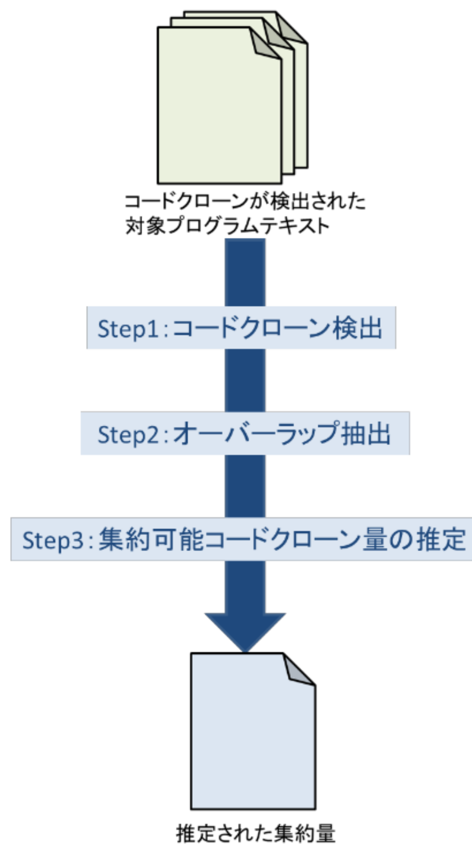

図 7: 提案手法の流れ

3　提案手法

本節では，集約可能コードクローン量を推定する手法について説明する．本研究では，メタヒューリスティクスの 4 つのアルゴリズムの中から最適なアルゴリズムを決定することを目的にしている．そのため，各アルゴリズムを適用して得られたコードクローンの集約可能コードクローン量を比較する．本手法では，比較を行うために必要なコードクローンの集約可能コードクローン量を求める．

本研究における，オーバーラップを含む，クローンセット分割を行ったコードクローンの集約可能コードクローン量の推定する手法の流れを説明する．

Step1　CCFinder のコードクローン検出結果を入力する．

Step2　入力したコードクローン間に発生したオーバーラップを抽出する．

Step3　メタヒューリスティクスなアルゴリズムを用いて，コードクローン全体の集約可能コー

ドクローン量を推定する．

図 7 には，本研究で提案する集約可能コードクローン量を推定する手法を示した．Step1 では，コードクローンの集約可能コードクローン量を推定するソースコードに対して，字句解析に基づく CCFinder によるコードクローンを検出を行い，その出力ファイルを入力に用いる．Step2 では，入力した位置情報を基に，どのコードクローン同士がオーバーラップしているかを判別する．Step3 では，検出されたすべてのコードクローンに対する，集約可能コードクローン量を推定する．また，無向グラフ G をもつクローンセットの集合に対して，各クローンセットを分割する．

コードクローンの検出

Step1 では，集約可能コードクローン量を推定するソースコードに対して，コードクローンを検出を行った．本手法では，字句単位の解析コードクローン検出ツール CCFinder を用いた．コードクローン検出に字句単位の解析ツールを用いる理由は，字句単位のような検出粒度が細かい検出法の方が集約可能なコードクローンを検出しやすいためである．メソッド単位やクラス単位の検出では集約可能なコードクローンは発生しにくい．また，CCFinder は様々な大規模ソフトウェアに適用され, その有用性が確認されている [6] ため，本手法で使用した．

3.1 オーバーラップの抽出

Step2 では，Step1 で検出したコードクローンをもとに，どのコードクローン同士がオーバーラップしているかを判別して，オーバーラップしているクローンセットを抽出する．同時に図 9 のような無向グラフ G を構成する，クローンセット S_1から S_6までを頂点とし，各頂点間に接続する無向辺に対して，隣接する 2 頂点はオーバーラップの関係にある．例えば，図 9 の場合，クローンセット S_1は 2 つのクローンセット S_2, S_3とオーバーラップしていることがわかる．

2 つのコード片 a_1, a_2が重複する条件は次の 2 つである．ただしコード片 a_1の開始行番号はコード片 a_2の開始行番号より先にあるものとする．

1. 2 つのコード片が同一ファイル内に存在する．
2. コード片 a_1の行番号 $\leq$ コード片 a_2の行番号 $\leq$ コード片 a_1の終了行番号を満たす.

以上, 2 つの条件を満たす 2 つのコード片 a_1, a_2はオーバーラップしている．また, 2 つのコード片 a_1, a_2に関して, $a_1 \in S_1, a_2 \in S_2$を満たすクローンセット S_1, S_2が存在するとする．このとき, $S_1 \neq S_2$ならば 2 つのクローンセット S_1, S_2はオーバーラップしている．ま

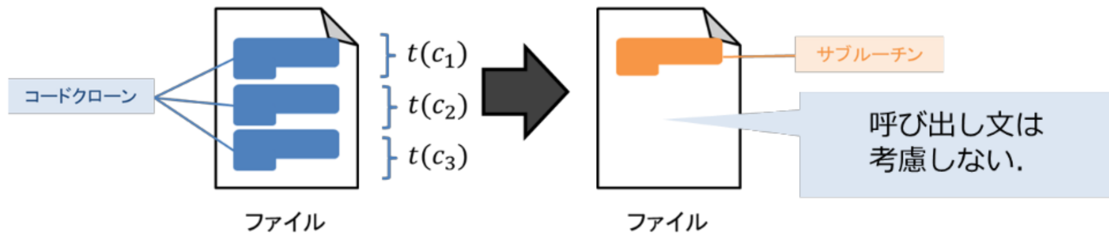

図 8: 集約可能コードクローン量の推定の概略図

た $S_1 = S_2$ならば，そのクローンセットは自分自身とオーバーラップしている. 無向グラフ G に属する任意のクローンセットは，無向グラフ G に属する他のクローンセットの少なくとも 1 つ以上とオーバーラップしている．

3.2 集約可能コードクローン量の推定

オーバーラップしているコードクローンに対して，集約可能コードクローン量を推定する場合，最大の集約可能コードクローン量を推定するためにはコードクローンを集約する順番を求めなくてはならない．理想的な解法としては，集約するコードクローンから構成される集合の全組み合わせに対して集約可能コードクローン量を推定して，最大の集約可能コードクローン量となる集合を解とすればよい．しかし，全組み合わせに対して，集約可能コードクローン量を推定するのは現実的ではない．本研究では，そのような問題に対して，メタヒューリスティクスを用いてコードクローンの集約可能コードクローン量の推定を行う．

本研究では，コードクローンの集約手法について特定の手法を選択しない．いずれの集約手法を選択しても，コードクローンであるコード片をプログラムに 1 つ残して，その他のコード片に関しては集約により消去されるという考え方を適用する．そのため，特定の集約手法に依存しない，コードクローンの集約可能コードクローン量を推定する．

集約可能コードクローン量を評価する指標として，ソースコードの行数やトークン数が候補として挙げられる．本研究では，開発者が集約の実行を率先して実行するかどうかを判断する基準としてコードクローンの集約可能コードクローン量を推定する．そのため，開発者が直感的に数値で評価しやすいと考えられるソースコードの行数を評価の指標とする．

任意のクローンセット S_kに関して, S_k内に含まれる集約可能なコードクローンの個数を n , i 番目のコードクローンを $c_{k,i}$とする．またコードクローン $c_{k,i}$のコード片の行数を $t(c_{k,i})$ と

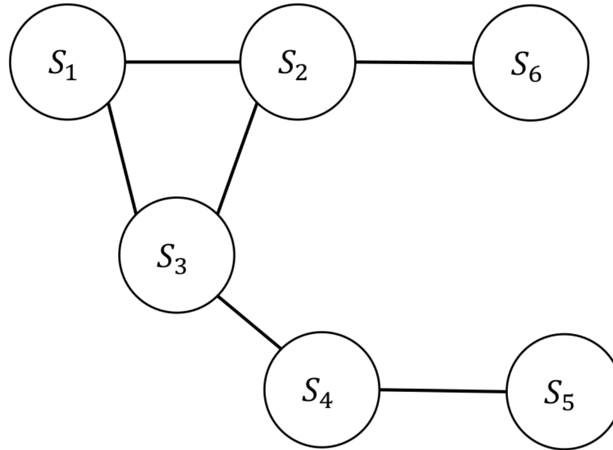

図 9: オーバーラップしているクローンセットが構成する無向グラフの例

する．

クローンセット S_kの集約を考える．コード片 1 つを残してその他のコード片を消去するのだから，クローンセット S_kの集約可能コードクローン量 $L(S_k)$ は次のように推定される．

$$L(S_k) = \frac{n-1}{n} \sum_{i=0}^{n} t(c_{k,i})$$

図 8 はサブルーチンに集約されるコードクローンの行数が $t(c_i)$ で与えられている様子を表している．ただし，本研究では，呼び出し文の行数を考慮していない．これは，集約可能コードクローン量が集約される行数だけではなく，集約されるトークン数を表すこともあるためである．

3.2.1 オーバーラップを含む集約可能コードクローン量の推定

複数のクローンセットのオーバーラップしている状態は図 9 のような無向グラフによって表現できる．クローンセット S_1から S_6までを頂点とし，各頂点間に接続する無向辺に対して，隣接する 2 頂点はオーバーラップの関係にある．例えば，図 9 の場合，クローンセット S_1は 2 つのクローンセット S_2, S_3とオーバーラップしていることがわかる．

図 9 のような無向グラフで隣接する複数のクローンセットからなる集合を G とする．また，集合 G に含まれる任意のクローンセットに関して，互いにオーバーラップしないようにクローンセットを選択してなす集合を H とする．すなわち，集合 H に含まれるクローン

セットに対する無向グラフでは，無向辺を持たない. 例えば，図 9 の場合，集合 G と，集合 H の一例として挙げる H_1, H_2は次のようになる．

$$G = \{S_1, S_2, S_3, S_4, S_5, S_6\}$$

$$H_1 = \{S_1, S_4, S_6\}$$

$$H_2 = \{S_3, S_5, S_6\}$$

2 つの集合 H_1, H_2に含まれるクローンセットは互いにオーバーラップしていないので，クローンセット S_kの集約可能コードクローン量を $L(S_k)$ とすると, 2 つの集合の集約可能コードクローン量 $L(H_1)$, $L(H_2)$ は次のようになる．

$$L(H_1) = L(S_1) + L(S_4) + L(S_6)$$

$$L(H_2) = L(S_3) + L(S_5) + L(S_6)$$

一般化すると，集合 H の集約可能コードクローン量 $L(H)$ は次のようになる．

$$L(H) = \sum_{S_i \in H} L(S_i)$$

集合 G の集約可能コードクローン量 $L(G)$ について, $L(G)$ の数値が最大となるようなクローンセットの選び方として集合 H_{max}が存在する．集合 H の集合を M とした場合，集合 G の集約可能コードクローン量 $L(G)$ は次のように表す.

$$L(G) = L(H_{max}) = \max_{H \in M}\{L(H)\}$$

3.2.2　クローンセットの分割

図 10 は，2 つのクローンセット A，B に対して，クローンセット B を優先的に集約した際に生じたクローンセット A に残されたコードクローンから構成されるクローンセット C が分割される様子を表した図である．集約可能コードクローン量はクローンセット単位で集約する想定で求めているが，オーバーラップはコード片単位で生じている．つまり，同じクローンセットに属するコードクローンのコード片が一様にオーバーラップしているわけではない．このような問題を解決するために，クローンセットの一部分を別のクローンセットとして分割することで，分割をしていないときに比べて集約可能コードクローン量を大きくすることができる．また自分自身とオーバーラップしているクローンセットの集約可能コードクローン量は，集約ができないので，0 であるが，分割を行うことでオーバーラップ解消がされて，集約可能になる可能性もある．

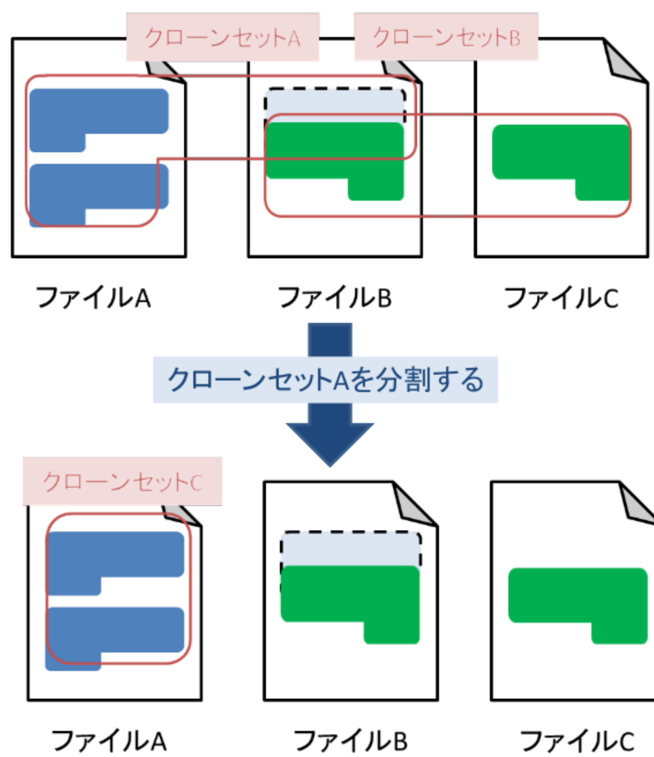

図 10: クローンセット分割の流れ

一般的に, n 個のクローンセットとオーバーラップしているクローンセット S_kの分割によってできる新しいクローンセットの個数は 2^n個である．しかし，実際には分割によって新しくできたクローンセットに関して，属するコードクローンの個数が 1 個以下であれば，そのクローンセットは集約可能コードクローン量が 0 であるために，消去しても問題がない．そのため，分割後に新しくできるクローンセットの個数は 2^n個より少なくなることがある．

3.3 探索的手法に基づくソフトウェア工学への適用

集約可能コードクローン量を決める集約の順番を求めるために，SBSE を用いる．そのために，Representation，Operators，Fitness Function を定義する必要がある．Step3 で，検出されたすべてのコードクローンに対する，集約可能コードクローン量を推定するアルゴリズムは Greedy，HC，SA，GA の 4 つである．

3.3.1 Representation

集約可能コードクローン量を推定する N 個のクローンセット全体に対して, i 番目のクローンセット S_kの状態 $S(i)$ を次のように表す．ただし, i は $N \geq k$ を満たす整数で，クローンセット同士を互いに識別するための ID である．

クローンセットの例： `S(i) = {Overlap(i)} = {p,q,r,…}`

$Overlap(i)$ は i 番目のクローンセット S_iとオーバーラップしているクローンセットの ID 集合を返す．例の場合, p, q, r 番目のクローンセットはそれぞれ i 番目のクローンセットとオーバーラップしている．このとき，クローンセット S_iは無向グラフ G に属しているものとする．ただし，無向グラフ G に属する任意のクローンセットは，無向グラフ G に属する他のクローンセットの少なくとも 1 つ以上とオーバーラップしている．無向グラフ G から任意のクローンセットを適当に選択し，それらのクローンセットを要素に持つクローンセットの新たな集合 H を考える．ただし，集合 H に含まれる任意のクローンセットは，集合 H に含まれるクローンセットとオーバーラップしていないものとする．

R は図 11 のようにバイナリ表現で考える．無向グラフ G に含まれる n 個のクローンセットをそれぞれ R で表されるバイナリの各位置に対応付ける．集合 H に含まれるクローンセットに対応する位置を 1，含まれないクローンセットに対応する位置には 0 を表示する．

Initial Representation

Greedy 法の初期値の決定は次の手順で行う．すべてのクローンセットを集約可能コードクローン量の大きい順番にソートして，集約可能コードクローン量が大きいクロー

1	2	3	4	…	n-1	n
1	0	0	1	…	0	1

図 11: Representation の例：n 個のクローンセット

ンセットから優先的に集約をしていく．ただしすでに集約したクローンセットとオーバーラップしていたために集約不可なクローンセットは集約しない．本研究におけるメタヒューリスティクスの初期値は Greedy 法で決定した集約の順番と同じとする．これは Greedy と比べて他のアルゴリズムの集約可能コードクローン量がどれだけ増加するのかを比べるためである．

3.3.2 Operators

集合 $H(k)$ に対する操作 Opr は次のように表せる．集合 $H(k)$ に対する，集合 H（$k+1$）の選び方はアルゴリズムに依存する．例えば，山登り法の場合，集合 $H(k+1)$ は集合 $H(k)$ の近傍である．または，遺伝的アルゴリズムの場合は，集合 $H(k+1)$ は集合 $H(k)$ を基に，生殖や自然淘汰，突然変異を起こすことで残る次の世代である．

$$Operators \text{ の例：} R(k+1) = Opr(R(k)) \quad = \quad Opr(H(k))$$

HC

$R(k)$ の近傍を選ぶアルゴリズムは以下のとおりである．

Step1 $R(k)$ の要素の中から任意の, 0 である i 番目の要素を選ぶ.

Step2 i 番目の要素を 1 にする.

Step3 i 番円の要素とオーバーラップしているクローンセットと対応している要素のうち, 1 である要素すべてを 0 にする.

Step4 $R(k)$ の要素の中から，集約しても他のクローンセットとオーバーラップしないクローンセットに対応する要素をすべて 1 にする.

Step5 得られた近傍の中から $FitnessFunction$ の評価に従い, $R(k+1)$ を返す.

$Step1$ で選択される要素は, $R(k)$ 中の要素 0 の数だけ存在する．また, $Step3$ の実行により，オーバーラップしているクローンセットは解消される. $Step4$ は $Step3$ によ

り 0 になった要素とオーバーラップしていたクローンセットが集約可能になっている可能性があるためである．ただし, 1 つのクローンセットの集約を解除することで 2 つ以上のクローンセットが集約可能になっていた場合, 2 つ以上のクローンセット同士がオーバーラップする可能性がある．このとき，クローンセット間でオーバーラップが発生しないようなクローンセットの全組み合わせを近傍として与える必要がある．

SA

$R(k)$ の近傍の選択は HC と同じでアルゴリズムを用いる．ただし $Step1$ に対して, SA は乱数を用いて，ただ 1 つの要素を選択する．また, $Step5$ に対して, SA では遷移確率というパラメータを用いる．遷移確率とは $R(k)$ が次の状態 $R(k+1)$ に遷移する際に，最適解に比べて評価値が離れた場合でも，より悪い状態を採択する確率である．本研究では，遷移確率は 0.1 から 0.9 までを 0.1 刻みで実行した. 4 章で示す SA の集約可能コードクローン量は，本実験では遷移確率による大きな差異はないためそれらの平均値としている．

GA

MOEA は Java のフレームワークで広く用いられるため，GA には多目的最適化問題に対して有効な，MOEA に搭載されている NSGAII アルゴリズムを使用する．変更可能なパラメータは評価回数，個体群数，交叉率，突然変異率の 4 つである．評価回数は個体群が世代交代を再現する回数のことである．本研究では，推定された集約可能コードクローン量が次第に増加していく様子がわかる 100 回から 2000 回の 100 回刻みで実行した．個体群数とは，1 世代に生存している個体群の数である．本研究では MOEA フレームワークのデフォルト値である 100 とした．次に，交叉率とは世代交代の際に個体群を交配するかしないかを決める確率のことである．本研究では，デフォルト値である 1.0 とした．最後に，突然変異とは個体群に含まれる個体を突然変異させる確立である．本研究では，デフォルト値である突然変異率は無向グラフ G に含まれるクローンセット数の逆数とした．個体群数，交叉率，突然変異率に対して，デフォルト値を使用した理由は，これらの値を動かしてもあまり結果に影響せず，代表としてデフォルト値を使用しても問題がなかったためである．

3.3.3 Fitness Function

集合 $H(k)$ に属するクローンセット全体の集約可能コードクローン量が大きいほど，より多く集約されたと見做せる．そのため，評価値は集合 $H(k)$ に属するクローンセット全体の集約可能コードクローン量とする．評価関数 $F(R(k))$ は，集合 H に属するクローンセット S_i

の集約可能コードクローン量の総和である．

$$FitnessFunction\,\text{の例：}\quad F(R(k)) = F(H(k)) = L(H(k)) = \sum_{S_i \in H} L(S_i)$$

4 適用実験

本章では，本研究の手法に適した，最大の集約可能コードクローン量を示すアルゴリズムが何であるのかを確認するために，適用実験を行った．

4.1 対象データ

本研究では，3 つのプロジェクト Apache Ant，ArgoUML，The Apache Xerces(以下 Xerces) のソースコードに含まれるコードクローンに対して集約可能量を推定した．3 つのプロジェクトは広く知られるオープンソースウェアで，コードクローンに関する研究の対象プロジェクトとしてよく用いられる [16]．各ソースコードの行数と，含まれるコードクローンが占める行数とオーバーラップしているコードクローンが占める行数，また，それらが占める割合を表 1，表 2，表 3 にそれぞれまとめた．最もコードクローンが占める行数が多いソースコードは ArgoUML で，最も少ないソースコードは Apache Ant である．また，表 4 には，各ソースコードで検出されたコードクローン，クローンセットの個数をまとめている．Apache Ant はコードクローン数，クローンセット数ともに 3 つの中で最小で，クローンセットを分割するとおよそ 2 倍ほどに増える．コードクローン数，クローンセット数ともに最大は ArgoUML で，クローンセットを分割するとおよそ 3 倍ほどに増える，Xerces も同様に，クローンセットを分割するとおよそ 3 倍ほどに増える．

4.2 各アルゴリズムによる集約可能コードクローン量推定結果の比較

表 5 から表 10 までに，各ソースコードに対する，メタヒューリスティクスを用いた集約可能コードクローン量推定結果の比較を示した．ただし，クローンセットの分割の有無による集約可能コードクローン量の差異も比較したいので，クローンセットの分割を行う場合と行わない場合の 2 通りの比較も行う．上の行から集約可能コードクローン量を推定するアルゴリズム，アルゴリズムに対応する集約可能コードクローン量 (行数)，ソースコードの全行数に対する集約可能コードクローン量の百分率，実行時間の順番でまとめている．SA，GA

表 1: Apache Ant ソースコードの内訳

	ソースコード	コードクローン	オーバーラップしている コードクローン
行数 (行)	265828	23278	13442
ソースコードに対する割合		8.76%	5.06%

表 2: ArgoUML ソースコードの内訳

	ソースコードの全行数	コードクローンの全行数	オーバーラップしているコードクローンの全行数
行数 (行)	389915	59787	41434
ソースコードに対する割合		15.33%	10.63%

表 3: Xerces ソースコードの内訳

	ソースコードの全行数	コードクローンの全行数	オーバーラップしているコードクローンの全行数
行数 (行)	238183	51731	36697
ソースコードに対する割合		21.72%	15.41%

表 4: 各ソースコードのコードクローンに関する内訳

	Apache Ant	ArgoUML	Xerces
コードクローン数	5785	21427	17249
クローンセット数 (分割無)	1740	4795	3792
クローンセット数 (分割有)	3552	15142	10341

は評価回数に応じて集約可能コードクローン量が変化するので，GA の評価回数 2,000 回を実行するのに要する時間とおよそ同じ実行時間だけ SA を実行して得られた集約可能コードクローン量を載せている．図 12 から図 17 までは，各ソースコードに対する，SA と GA において評価回数ごとの集約可能コードクローン量の推移を，クローンセットの分割の有無で分けて比較している．水平軸が評価回数で，垂直軸が集約可能コードクローン量 (行) である．評価回数は集約可能コードクローン量の推移を観察するために，最小を 100 回，最大を 2,000 回として 100 回ずつ刻んで実行した．

Apache Ant

表 5 は Apache Ant に対する，クローンセットを分割しない場合のコードクローンの集約可能コードクローン量を比較する表である．また，表 6 は Apache Ant に対する，クローンセットを分割をした場合のコードクローンの集約可能コードクローン量を比較する表である．最大の集約可能コードクローン量を示したのは GA である．また，2 つの表ともに HC と SA を比較した場合，実行時間は SA が短く，集約可能コードクローン量は SA が多いことがわかる．Apache Ant では分割の有無で集約可能コードクローン量の変化がみられるが，後に見る 2 つのソースコードに比べると差が小さい．また，各アルゴリズム間でも，集約可能コードクローン量の差はかなり小さい．SA と GA で得られる集約可能コードクローン量の百分率を比較した場合，0.01%以下の僅差である．図 12 は，Apache Ant に対して，クローンセット分割無しでの，SA と GA の集約可能コードクローン量の推移を示している．図 13 は，Apache Ant に対して，クローンセット分割有の場合である．GA は SA を上回る集約可能コードクローン量を示すことが多かった．図 13 において，少ない評価回数の間は SA が GA より上回っているが，すぐに逆転して GA が SA を上回っている様子がわかる．

ArgoUML

表 7 は ArgoUML に対する，クローンセットを分割しない場合のコードクローンの集約可能コードクローン量を比較する表である．また，表 8 は ArgoUML に対する，ク

表 5: Apache Ant に対するコードクローンの集約：分割無

	Greedy	HC	SA	GA
集約可能コードクローン量	10960	11093	11162	11170
ソースコードに対する割合	4.12%	4.17%	4.20%	4.20%
実行時間 (s)	0.015	0.17	10.22	16.51

表 6: Apache Ant に対するコードクローンの集約：分割有

	Greedy	HC	SA	GA
集約可能コードクローン量	11918	12289	12427	12448
ソースコードに対する割合	4.48%	4.62%	4.67%	4.68%
実行時間 (s)	0.031	0.125	15.35	10.82

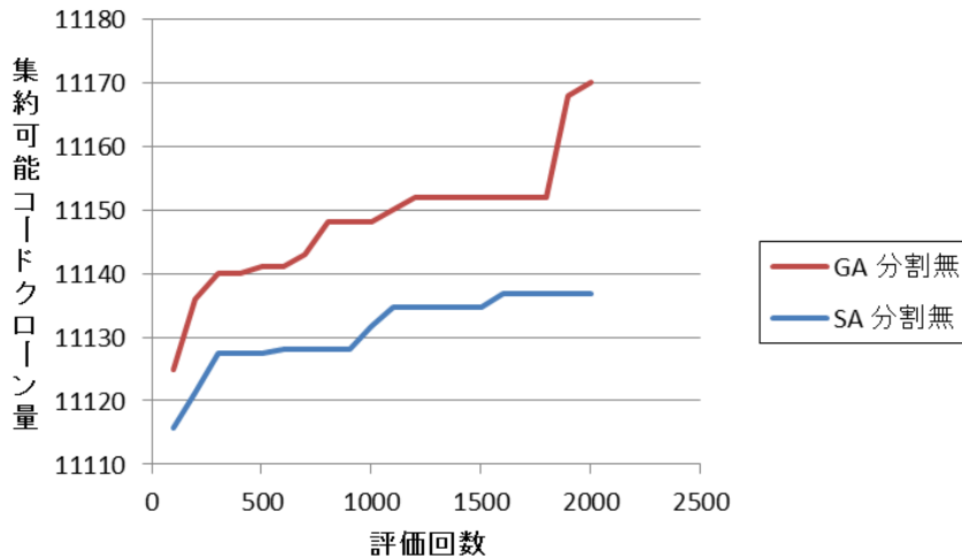

図 12: Apache Ant に対する，クローンセット分割を行わない時の SA と GA の集約可能コードクローン量の評価回数に関する推移

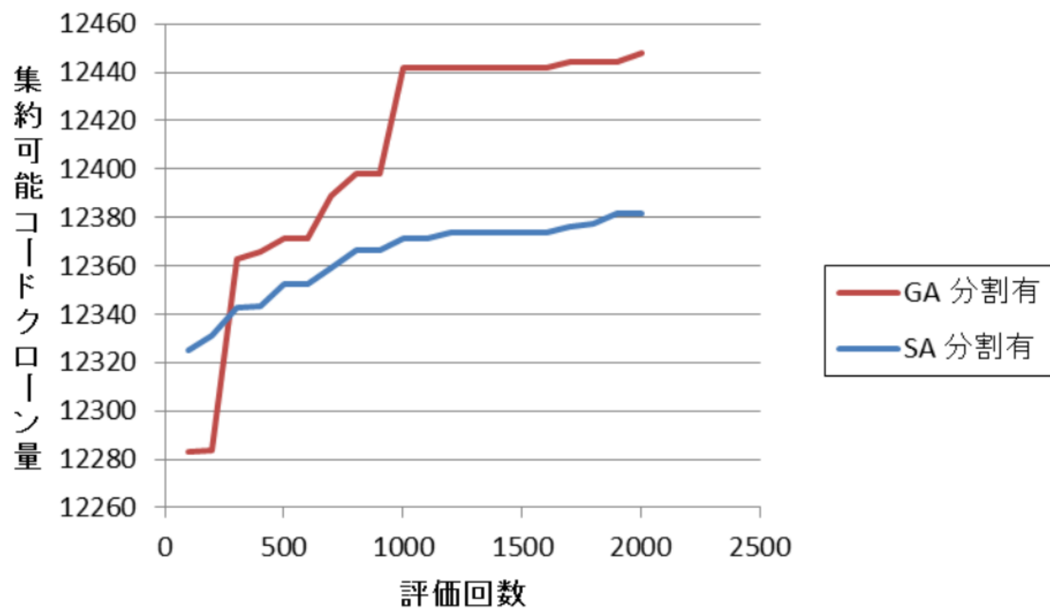

図 13: Apache Ant に対する，クローンセット分割を行う時の SA と GA の集約可能コードクローン量の評価回数に関する推移

ローンセットの分割をした場合のコードクローンの集約可能コードクローン量を比較する表である．クローンセットの分割無の集約可能コードクローン量に比べて，分割有の集約可能コードクローン量は 1.3%以上増加して，特に GA は 9%に達している．また，クローンセットの分割有の場合，Greedy に比べて，他のメタヒューリスティクスの集約可能コードクローン量は 1,700 行を超える差がある．また，HC より SA や GA の方が集約可能コードクローン量は優れているのがわかる．．図 14 は，ArgoUML に対して，クローンセット分割無しでの，SA と GA の集約可能コードクローン量の推移を示している．図 15 は argoUML に対して，クローンセット分割有の場合である．どちらの図でも，評価回数が少ない間は SA が GA を上回る集約可能コードクローン量である．GA は評価回数が増えるごとに集約可能コードクローン量を増加させ，SA を上回る．

Xerces

表 9 は Xerces に対する，クローンセットを分割しない場合のコードクローンの集約可能コードクローン量を比較する表である．また，表 10 は Xerces に対する，クローンセットを分割をした場合のコードクローンの集約可能コードクローン量を比較する表である．分割の有無による集約可能コードクローン量の差は，2 つの表を見比べるとおよそ 3,000 行である．割合では 1.2%以上の変化がある．ArgoUML の場合と同様に，クローンセットを分割したことによる集約可能コードクローン量への影響がわかる．また，Greedy に比べて他のメタヒューリスティクスを用いた場合の集約可能コードクローン量は 1,500 行以上の差がある．また，3 つのメタヒューリスティクスを比べて場合，GA が最大の集約可能コードクローン量を示している．図 16 は，Xerces に対して，クローンセット分割無しでの，SA と GA の集約可能コードクローン量の推移を示している．図 17 は，Xerces に対して，クローンセット分割有の場合である．クローンセット分割がない場合，SA の集約可能コードクローン量が GA の集約可能コードクローン量を常に上回っていた．また，クローンセット分割がある場合，評価回数が少ない間は SA が GA をやや上回っていたが，評価回数が増加するにつれて，GA は

表 7: ArgoUML に対するコードクローンの集約：分割無

	Greedy	HC	SA	GA
集約可能コードクローン量	28447	29288	29491	29460
ソースコードに対する割合	7.30%	7.51%	7.56%	7.56%
実行時間 (s)	0.095	1.139	24.18	23.51

表 8: ArgoUML に対するコードクローンの集約：分割有

	Greedy	HC	SA	GA
集約可能コードクローン量	33084	34705	35028	35235
ソースコードに対する割合	8.48%	8.90%	8.98%	9.03%
実行時間 (s)	0.095	1.514	33.91	29.20

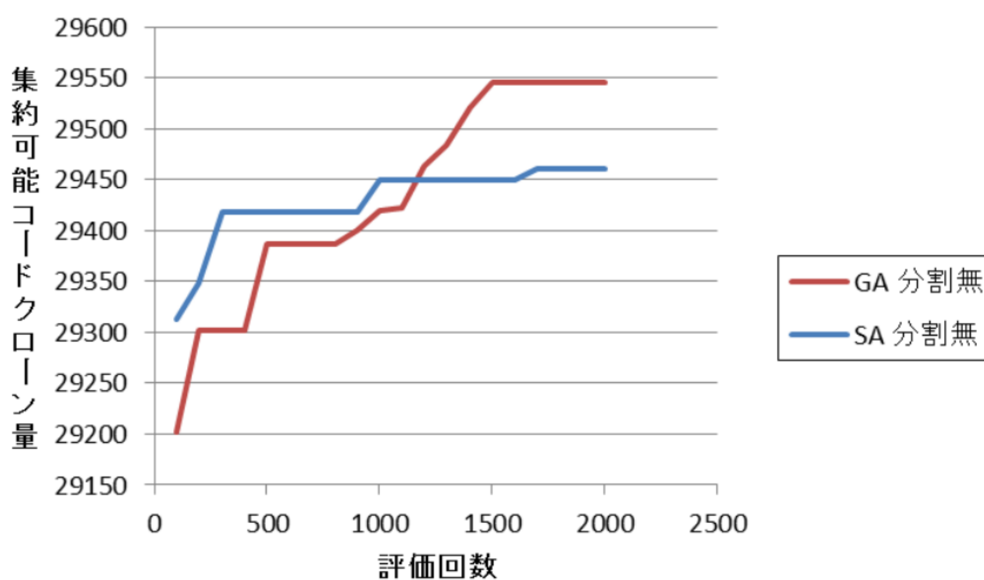

図 14: ArgoUML に対する，クローンセット分割を行わない時の SA と GA の集約可能コードクローン量の評価回数に関する推移

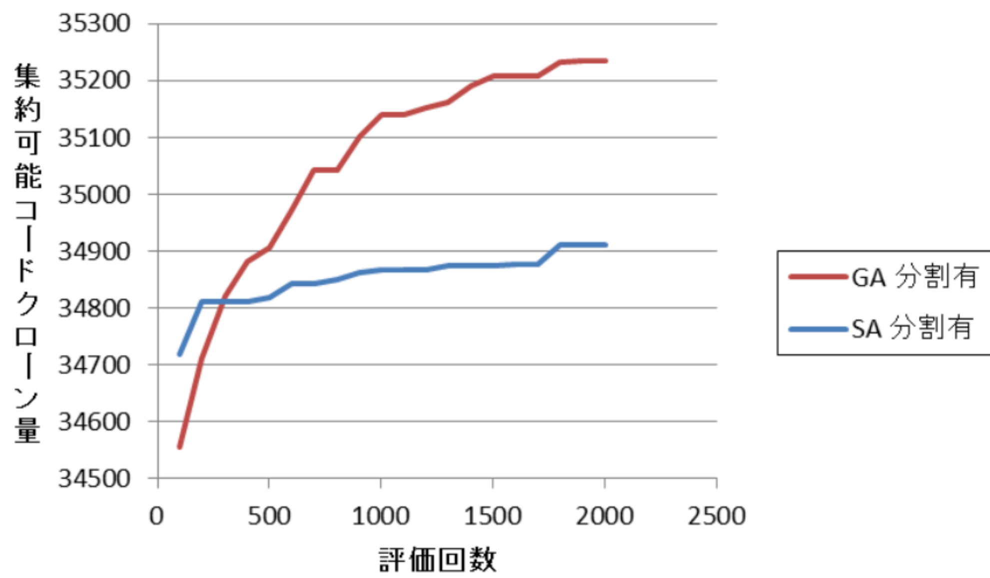

図 15: ArgoUML に対する，クローンセット分割を行う時の SA と GA の集約可能コードクローン量の評価回数に関する推移

SA を上回る集約可能コードクローン量を示した．

4.3　クローンセットの分割に要する実行時間

図 11 は，各ソースコードに対する，クローンセットを分割するのに必要な時間をまとめている．単位はミリ秒である．最も早かったのは，Apache Ant でおよそ 4 秒ほどだった．次いで早かったのは Xerces でおよそ 1 分 46 秒ほどだった．最も時間が必要だったソースコードは ArgoUML で，およそ 36 分必要だった．ArgoUML では，分割前のクローンセットの数が 4,795 で，分割後には 15,142 になっている．2 万以上のコードクローンを持っていたので，これらが複雑にオーバーラップしていた場合，大きな時間が必要だったと考えられる．

4.4　実行時間あたりの集約可能コードクローン量の比較

4.2 節より，評価回数あたりでは GA の方が SA よりも集約可能コードクローン量が上回る．この節では，補足として SA と GA の実行時間あたりの集約可能コードクローン量を求める性能を比較する．図 18，図 19，図 20 は水平軸を実行時間 (s)，垂直軸を集約可能コードクローン量とした線グラフである．

Apache Ant

図 18 は Apache Ant の集約可能コードクローン量を推定して得られた，SA と GA の実行時間に対する集約可能コードクローン量の線グラフである．SA の集約可能コードクローン量は 5 秒前後を境に，GA の集約可能コードクローン量を超えられていない様子がわかる．

ArgoUML

図 19 は ArgoUML の集約可能コードクローン量を推定して得られた，SA と GA の実行時間に対する集約可能コードクローン量の線グラフである．SA の集約可能コードクローン量は 10 秒前後を境に，GA の集約可能コードクローン量を超えられていない様子がわかる．

表 9: Xerces に対するコードクローンの集約：分割無

	Greedy	HC	SA	GA
集約可能コードクローン量	23554	24368	24603	24500
ソースコードに対する割合	9.89%	10.23%	10.32%	10.28%
実行時間 (s)	0.047	0.061	12.11	11.28

表 10: Xerces に対するコードクローンの集約：分割有

	Greedy	HC	SA	GA
集約可能コードクローン量	25852	27395	27806	27870
ソースコードに対する割合	10.85%	11.50%	11.67%	11.70%
実行時間 (s)	0.033	0.75	15.14	14.0

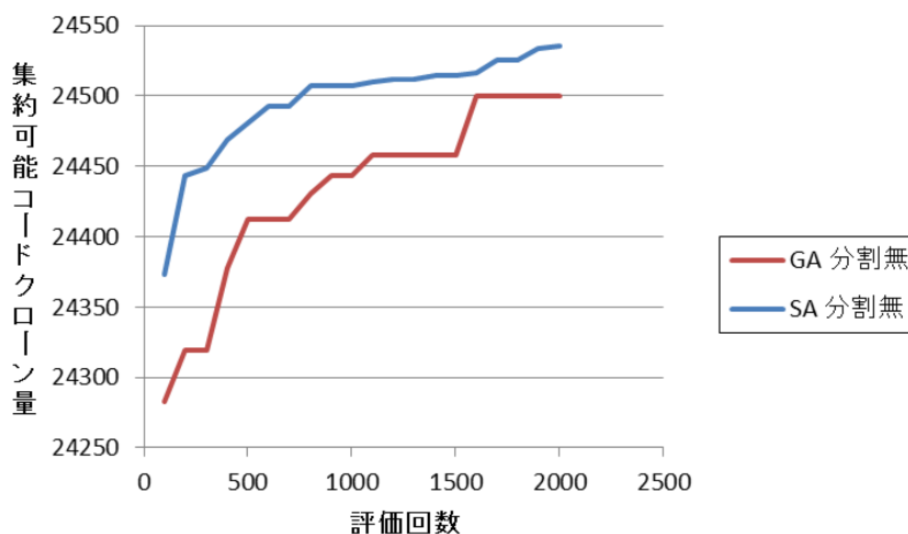

図 16: Xerces に対する，クローンセット分割を行わない時の SA と GA の集約可能コードクローン量の評価回数に関する推移

表 11: 各ソースコードに対する，クローンセットを分割するのに要する時間 (s)

	Apache Ant	ArgoUML	Xerces
分割時間 (s)	4.04	216.44	106.47

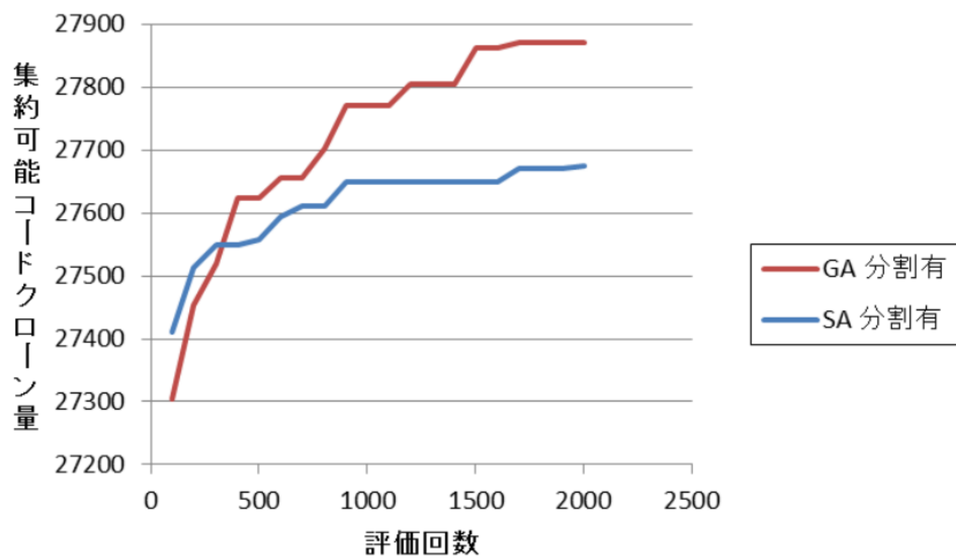

図 17: Xerces に対する，クローンセット分割を行う時の SA と GA の集約可能コードクローン量の評価回数に関する推移

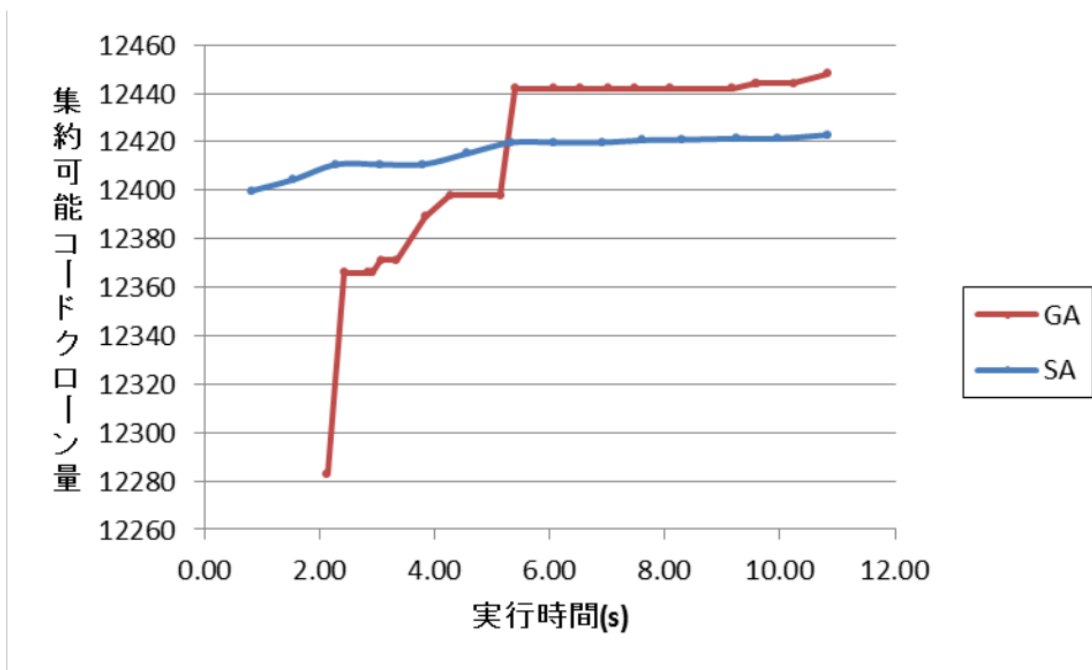

図 18: Apache Ant に含まれるコードクローンの集約可能コードクローン量を推定して得られた，SA と GA の，実行時間に対する集約可能コードクローン量

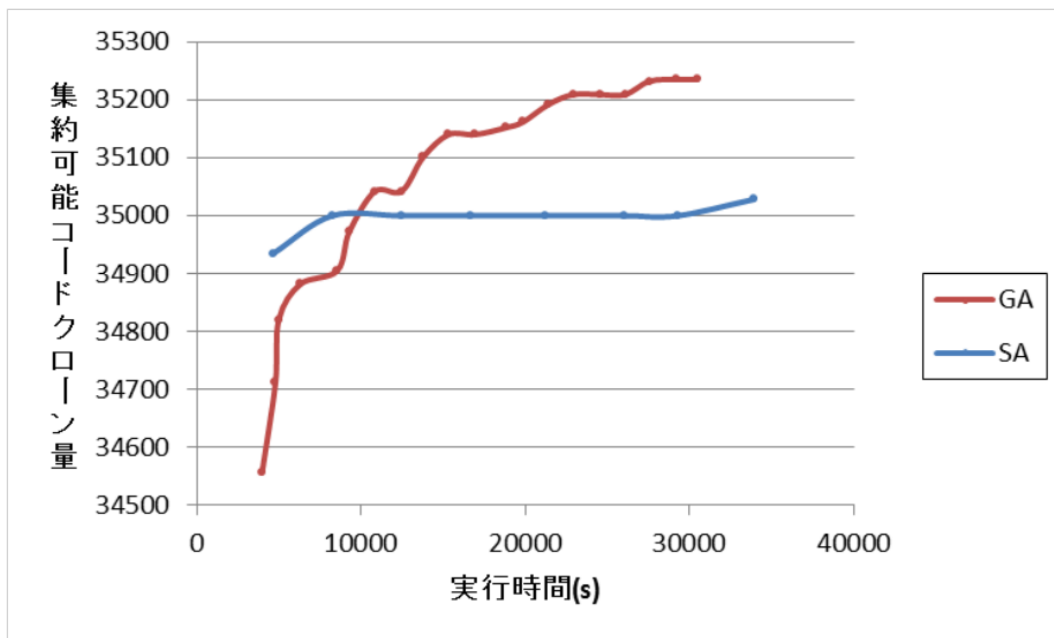

図 19: ArgoUML に含まれるコードクローンの集約可能コードクローン量を推定して得られた，SA と GA の，実行時間に対する集約可能コードクローン量

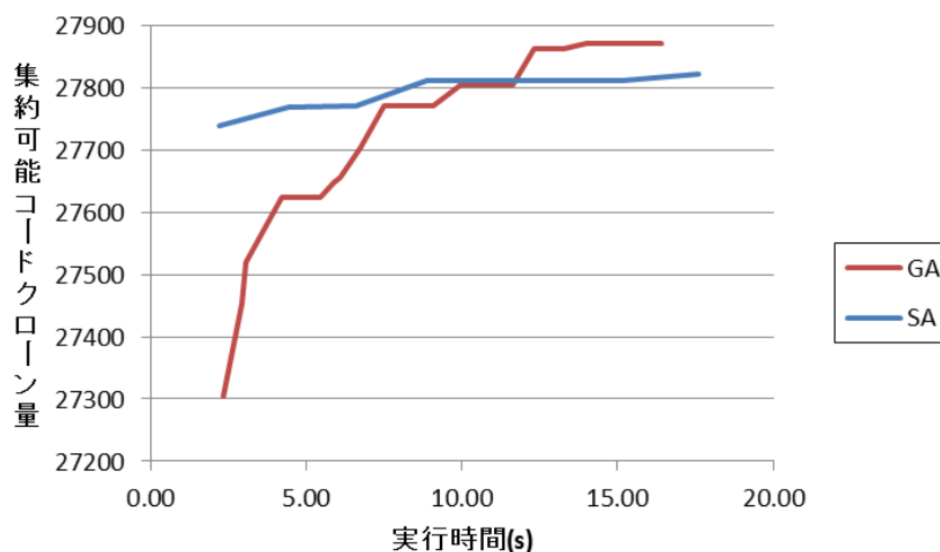

図 20: Xerces に含まれるコードクローンの集約可能コードクローン量を推定して得られた，SA と GA の，実行時間に対する集約可能コードクローン量

Xerces

図 20 は Apache Ant の集約可能コードクローン量を推定して得られた，SA と GA の実行時間に対する集約可能コードクローン量の線グラフである．8 秒当たりから SA と GA の集約可能コードクローン量はおおよそ等しくなる．また，20 秒に近づくにつれ，GA の集約可能コードクローン量が SA を上回り始めている．

4.5 考察

クローンセット分割による集約可能コードクローン量の変化

表 5 から表 10 に関して，それぞれのソースコード毎にクローンセットの分割をしていない場合とした場合を比較する．いずれの場合もクローンセットを分割したことによる影響がある．特にコードクローンの量が多く，更にオーバーラップが多いほど，クローンセットを分割した際に集約可能コードクローン量が増加する傾向が見られた．

Apache Ant は 3 つの中では，コードクローンの量が最も少なく，クローンセットを分割しても集約可能コードクローン量の増加は小さかった．ArgoUML は 3 つの中では，最大のコードクローン量であり，オーバーラップも豊富にあり，クローンセット分割による集約可能コードクローン量の増加も最大であった．

分割に要する実行時間の改善

表 11 には，3 つのソースコードに対する，クローンセット分割を行ったときの実行時間を表にまとめたものである．argoUML のように，コードクローンの数が多く，オーバーラップも多ければ，クローンセットの分割により新たにできるクローンセットの数は多くなる．n 個のクローンセットとオーバーラップしているクローンセットを分割してできる新たなクローンセットは 2^n と指数関数的に増加する．今回の推定で使用した 3 つのソースコードは，とりわけ大きいサイズではない．より大きなサイズを持つソースコードから検出されたクローンセットを分割した際に，分割を実行するために必要な時間が開発者にとって不利になる可能性がある．そのため，クローンセット 1 つあたりのオーバーラップが多いクローンセットの分割の手法に関して，実行時間を短縮する必要がある．

アルゴリズムの比較：Greedy Algorithm

Greedy アルゴリズムは他 3 つのアルゴリズムに比べて，集約可能コードクローン量が最小で実行時間が最短であった．また，実装がもっとも単純なアルゴリズムである．コードクローン間でオーバーラップを発生していない場合の集約可能コードクローン量は Greedy で得られる集約可能コードクローン量に一致する．そのため，対象のソースコードから検出したコードクローンの数やオーバーラップの発生が少ない場合には，最適なアルゴリズムだと考えられる．反対に，ArgoUML のようにオーバーラップしているコードクローンが多いと，Greedy で推定される集約可能コードクローン量の誤差は大きくなる．

アルゴリズムの比較：HC

HC アルゴリズムは Greedy アルゴリズムより推定される集約可能コードクローン量が多く，しかし SA や GA に比べると少ないアルゴリズムである．HC は評価関数が凸関数にできれば大きな効果を持つ，局所的な探索が得意なアルゴリズムである．しかし，極大値にはまると，それを大域的な最大値として推定してしまう弱点がある．本研究のようなクローンセットを集約する順番を 1 つ変えるだけで，評価値が大きく変動してしまう問題に対して，あまり良い結果を出すことができなかったと考られる．

アルゴリズムの比較：SA

SA アルゴリズムは評価回数を基準にした場合，GA に次いで集約可能コードクローン量の大きいアルゴリズムである．集約可能コードクローン量と評価回数の関係を表した図 12 から図 17 より，評価回数の増加に伴い，集約可能コードクローン量は微増する．図 18 から図 20 より，GA の実行時間より多くしても GA の集約可能コードクローン量を上回らなかった．HC と同様に，SA は極大値にはまると，それを大域的な最大値として推定してしまう弱点がある．そのため，あまり良い結果を出すことができなかったと考えられる．

アルゴリズムの比較：GA

GA アルゴリズムは本研究で最大の集約可能コードクローン量を示したアルゴリズムである．また，4.2 節，4.3 節，4.4 節で SA と比較をして，実行時間当たりの集約可能コードクローン量で SA よりも集約可能コードクローン量が優れていた．このことは，本研究の集約可能コードクローン量の推定にメタヒューリスティクスを適用した場合，局所的な探索手法よりも大局的な探索手法が優れていることを示した．

5　まとめと今後の課題

本研究では，オーバーラップを考慮したコードクローンの集約可能コードクローン量を推定するために，探索的手法に基づくソフトウェア工学を利用した．クローンセットを分割することでオーバーラップへの有効性があることを確かめた．また，HC や SA，GA などメタヒューリスティクスを，実行時間やコードクローン集約可能コードクローン量を対象に性能を比較した．最大の集約可能コードクローン量を示したのは GA で，次いで SA だった．つまり，コードクローンの集約可能コードクローン量を求める有効な手段としては，局所的な探索的手法よりも大域的な，個体群に基づいた探索的手法の方が適切だと考えられる．

今後の課題として，クローンセット分割の手法に関する実行時間の改善が挙げられる．開発者がコードクローンの集約可能コードクローン量を推定する時に，ArgoUML よりも大規模なソフトウェアでは実行するのに多くの時間を要する可能性がある．また，本研究で対象にしたソースコードから検出されたコードクローンが一般的なソフトウェアのコードクローン量に比べて小さいため，より大規模なソフトウェアを対象にした実験が必要になる．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上克郎 教授には, 御多忙の中，常に適切な御指導及び御助言を賜りました. また，研究室の環境が研究に大変専念させていただきやすかったこと，深く感謝を申し上げます.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授には進捗報告を通じて，研究について多くの御助言を頂きました. また，研究だけではなく学業に関しましても多くの御助言をいただきましたことを，心より感謝します.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教には中間報告会などを通じて，研究について貴重なご意見を頂きました. また，今後の研究でも重宝する御助言をいただきましたこと，心より感謝します.

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 Ali Ouni 特任助教には本研究で使用した MOEA ツールの使い方を教えて頂きました．また，研究の進捗の様々な場面で多くの御助言をいただきましたこと，心より感謝します.

名古屋大学大学院情報科学研究科附属組込みシステム研究センター / 情報システム学専攻 吉田則裕 准教授には本研究について多くの御指導をしていただきました．また，本論文を執筆にあたり数々の御助言を賜りましたことを，深く感謝しています.

大阪大学大学院国際公共政策研究科 崔 恩瀞 助教，大阪大学大学院情報科学研究科コンピュータサイエンス専攻　中村 勇太氏には研究の御支援および本論文の修正，推敲などご尽力いただきました．また，学生生活の中でも多くの御助言をいただいたことを感謝しています.

最後に，様々な御指導，御助言等を頂き, 研究生活を支えてくださった, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, Markus Neteler. A novel approach to optimize clone refactoring activity. *Proceedings of the 8th annual conference on Genetic and evolutionary computation GECCO '06*, pp. 1885–1892, 2006.

[2] 欲張り法(greedy strategy). http://www.geocities.jp/m_hiroi/light/pyalgo22.html(2016年2月16日).

[3] *Genetic Algorithm Research Group / Intellingent Systems Designs Lab .,Doshisya Univ.* http://mikilab.doshisha.ac.jp/dia/research/pdga/research.html(2016年1月31日).

[4] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. *Empirical Software Engineering and Verification*, pp. 1–59, 2012.

[5] Algorithms with Python ヒューリスティック探索. http://moeaframework.org/(2016年2月16日).

[6] 肥後芳樹, 楠本真二, 井上克郎. コードクローン分析ツール gemini を用いたコードクローン分析手法. 電子情報通信学会, Vol. 105, No. 228, pp. 37–42, 2005.

[7] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[8] 肥後芳樹, 吉田則裕. コードクローンを対象としたリファクタリング. コンピュータソフトウェア, Vol. 28, No. 4, pp. 43–56, 2011.

[9] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001.

[10] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. *Proceedings of the 29th international conference on Software Engineering ICSE '07*, pp. 96–105, 2007.

[11] 神谷年洋, 肥後芳樹, 吉田則裕. コードクローン検出技術の展開. コンピュータソフトウェア, Vol. 28, No. 3, pp. 29–42, 2011.

[12] Heejung Kim, Yungbum Jung, Sunghun Kim, Kwankeun Yi. Mecc: memory comparison-based clone detector. *Proceedings of the 33rd International Conference on Software Engineering ICSE '11*, pp. 301–310, 2011.

[13] *MOEA Framework:A Free and Open Source Java Framework for Multiobjective Optimization.* http://www.geocities.jp/m_hiroi/light/pyalgo28.html(2016 年 2 月 16 日).

[14] Mark O'Keeffe, Mel O Cinneide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice - Search Based Software Engineering [SBSE]*, Vol. 20, No. 5, pp. 345–364, 2008.

[15] Simulated Annealing/SA-概論-. http://mikilab.doshisha.ac.jp/dia/research/person/wako/sa_intro1.htm(2015 年 1 月 31 日).

[16] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, 2014.