ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

北 椋太† 岡田 龍太郎† 峰松 彩子† 中西 崇文†

†武蔵野大学データサイエンス学部データサイエンス学科 〒135-8181 東京都江東区有明 3-3-3

E-mail: s2022007@stu.musashino-u.ac.jp,
{ryotaro.okada, ayako.minematsu, takafumi.nakanishi}@ds.musashino-u.ac.jp

あらまし 本稿では，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．一般的に，プログラミングにおいて，構造を簡略化することは，生産性やのちの保守性の向上において重要であり，これを自動化することは，プログラマにとって労力の軽減につながる．構造を簡略化する方法の一つとして，コードクローンを関数としてまとめることが挙げられる．本方式は，任意のソースコードを入力として，そのソースコードを構文木として表現した上で，各部分木間の類似度を計量することにより，コードクローンを検出し，新たに生成した関数に置き換える．本方式を実現することで，より簡略化した保守性の高いソースコードを書くことが可能になる．

キーワード ソースコード, コードクローン，構文木, 構造類似性, 関数生成

1. はじめに

近年，ソフトウェアの利用分野の拡大にともない，プログラムの不具合やそれにともなうシステムの故障が社会的に問題となることが多くなっている．ソフトウェア開発ライフサイクル全体において，保守作業のコストが占める割合は非常に高く，プロジェクトの大規模化にともなって，その割合はさらに増加する．そこで，ソフトウェアの保守作業の効率化が重要な課題となっている．ソフトウェアの開発や，保守作業を行うにあたり，開発者は，ソースコードからプログラムの構造を理解し，修正する必要があり，この作業がソフトウェアのメンテナンスの多くの割合を占めている．そのため，ソフトウェアの保守作業に要するコストを削減するためには，ソースコードの理解や，修正を容易にすることが重要である．これらの作業を困難にする要因の一つとして，コードクローンが挙げられる．コードクローンとは，ソースコード中の一部分(コード片)のうち，類似または一致したコード片がソースコード中に複数存在するもののことである．あるコード片にて欠陥が発見された場合には，そのコード片の全てのコードクローンを探し，検査や修正を検討する必要がある．しかし，大規模なソフトウェア開発において．ソースコード中のコードクローンを手作業で探し，修正を行うには，非常に大きな労力が必要となる．一般にコードクローンを検索する方法としてキーワード検索や，コードクローン検出ツールが挙げられる．キーワード検索では，欠陥を含むコード片から抽出したキーワードから，grep[1]などを用いて検索する．しかし，キーワード検索では，キーワードと完全に一致するコード片を出力するため，識別子の違いがあるなどのわずかに異なるコードクローンは検出することができない．また，効率的な検索を行うための適切なキーワードを選定するためには，対象となるソースコードを十分に理解している必要がある．一方，コードクローン検出ツールを用いる場合，欠陥を含むコード片を与えることで，等価なコード片の一覧や位置情報が表示される．しかし，これらの手法では，検出されたコードクローンに対して，手作業で修正を行う必要がある．

本稿では，ソースコードを構文木で表現することで，構文木内の類似した構造を含むコード片を検出し，抽出したコードクローンを再利用可能な関数に置き換える手法を提案する．

本方式では，構文木内の構造に基づいてコードクローンを検出することにより，識別子の違いなどの処理に影響を及ぼさない差異を無視したコードクローンの検索を可能にする．また，検出したコードクローンを関数化することにより，コードクローンの解消を実現する．

本稿の構成は，次の通りである．2 章では，関連研究について紹介する．3 章では，本提案方式であるソースコードの構文木表現による構造類似性を用いた自動関数化方式について示す．4 章では，本方式を実現するシステムを構築し，実験を行う．5 章で本稿をまとめる．

2. 関連研究

本章では，本方式に関連する研究について述べる．

本節の構成は，次の通りである．2.1 節では，コードクローン検出に関する研究について述べる．2.2 節では，ソースコードの簡略化に関する研究について述べる．

2.1. コードクローン検出に関する研究

コードクローン検出の研究は，ソースコードをさまざまなモデルで表現することにより実現されてきた[2]．

本節では，モデルとして主にテキストやトークンの並びで表現する字句を用いた検出手法と，構文木やグラフで表現する構文を用いた検出手法の二つに分類し，関連する研究について述べる．

2.1.1. 字句を用いた検出手法に関する研究

神谷ら[3]は，プログラムテキストをトークンの並びで表現し，そのトークン列の等価性に基づいてコードクローンを検出する CCFinder を開発した．

吉田ら[4]は，コード片に含まれる識別子の類似性に基づいてコードクローンを検出する手法を提案している．この研究では，ソースコード内の関数に含まれる語を抽出し，共起関係に基づいて類義語を特定し，入力コード片が含む語と一致する，もしくは類似する語を含む関数を提示している．

2.1.2. 構文を用いた検出手法に関する研究

Gabel ら[5]は，ソースコードから program dependence graph(PDG)の形で構造化することにより，コード片の検出を部分グラフの類似度計量の問題にすることでコードクローンを検出する手法を提案している．

Jiang ら[6]は，ソースコードを抽象構文木で表現し，木構造の類似度を算出することでコードクローンを検出する DECKARD を開発した．

2.2. ソースコードの簡略化に関する研究

石原ら[7]は，ファイル単位にまたがって存在するコードクローンのライブラリ化を目的に，大規模なソフトウェア群を対象に，メソッド単位のコードクローンを検出する手法を提案している．

Baxter ら[8]は，抽象構文木を用いたコードクローンの検出手法を提案する中で，部分木の類似度に対して任意の閾値を設定し，閾値以上の類似度を持つコードクローンをマクロとしてまとめ，それを更に再帰的に繰り返すことで，より大きな構造に対してコードクローンを検出することを可能にした．

2.3. 本研究の位置付け

本研究では，入力されたソースコードを構文解析することで，抽象構文木で表現する．抽象構文木内の各ノードには，親子関係およびソースコード内での行番号などの情報が保持されている．本研究では，さらに，各ノードに，そのノードから見た子孫ノードの葉との最大距離を保持している．この情報を用いて，任意の深さ以上の部分木を全て抽出したのちに，一致または類似する構造を含む部分木の組み合わせを検出する．検出した部分木に該当するコード片をコードクローンとみなし，これを解消するため，関数の生成を行う．

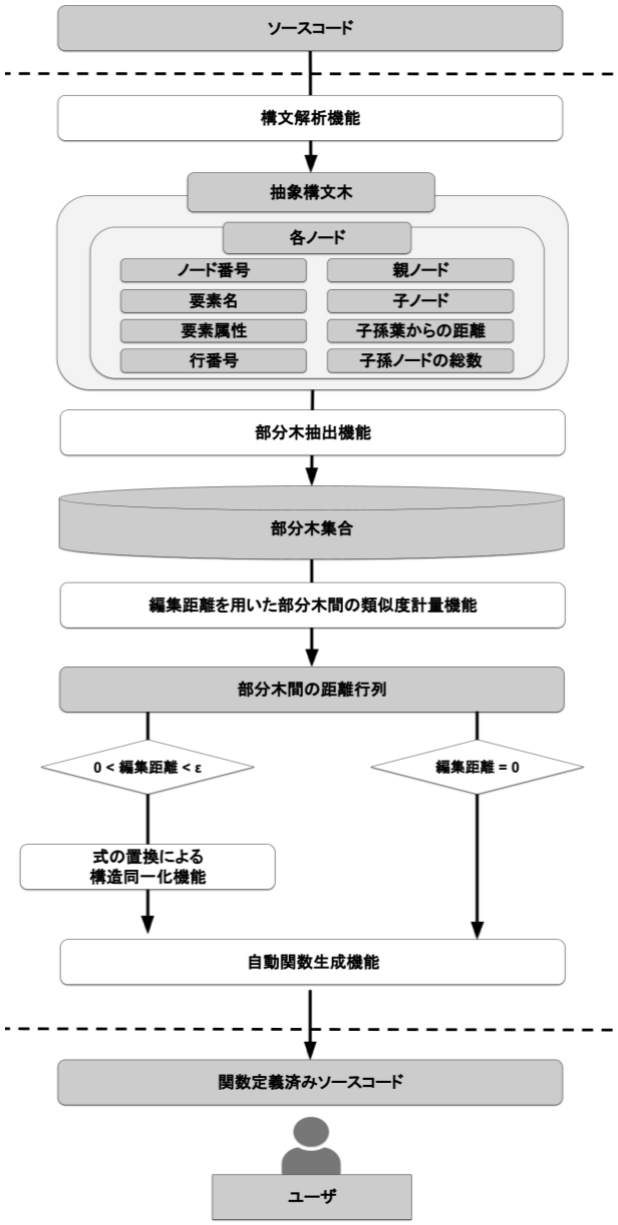

図 1 提案システムの全体像

これまでのコードクローン検出に関する研究では，類似するコード片を検出し，位置情報または抽出したコード片を出力とすることが主であった．そのため，あるコード片にて不具合が発見された場合には，コードクローンの検索は効率化されるものの，最終的には手作業で修正を行う必要がある．一方，本研究では，抽象構文木を用いて検出したコードクローンを新たに生成した関数に置き換えることで，コードクローンを解消することが可能となる．また，本研究では，抽出

する部分木の深さをパラメータとして任意の値で設定することで，コードクローンを検出するスコープを変更することができる．

3. ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

本章では，提案方式である，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について示す．

3.1. 提案手法の概要

本節では，提案手法の全体像について述べる．提案システムの全体像を図 1 に示す．

本方式では，入力データとして任意のソースコードを与え，構文解析を行い，抽象構文木を生成する．生成した抽象構文木から，指定した深さパラメータ以上の深さである部分木を全て抽出し，部分木の集合を作成する．そして，部分木間の類似度を計量することでコードクローン検出する．その部分木間の類似度計量手法として，構文木の編集距離による類似度計量手法を用いる．一致する部分木に対して関数の生成を行う．また，一致でない部分木においても，式に対応する部分木を置換することによって構造を同一化し，関数生成を適用する範囲を拡張することが可能である．式とは，プログラムの部分構造であり，評価後に値に変換されるものである．例えば，変数やリテラル，それを組み合わせた演算子式などが式にあたる．

本手法は，構文解析機能，部分木抽出機能，編集距離を用いた部分木間の類似度計量機能，式の置換による構造同一化機能，自動関数生成機能からなる．

3.2. 構文解析機能

本節では，ソースコードから抽象構文木を生成する構文解析について述べる．構文解析は，ソースコードから要素の最小単位であるトークンを取り出す字句解析を行なったのち，そのトークン間の関係を明確にする手続きのことである．この際，その結果を木構造で表現したものが構文木であり，特にトークンの詳細な情報を取り除き，動作の意味に関係ある情報のみを取り出した構文木を抽象構文木と呼ぶ．本システムでは，変数などの動作に関与しない要素を省略するために，抽象構文木を用いることで構造のみから類似度を計算する．本方式では，構文木からソースコードへ再変換するために，各ノードはソースコードの行番号および列番号をノードに保持する．

3.3. 部分木抽出機能

本節では，抽象構文木から部分木を抽出する方法について述べる．関数生成を行うにあたって，小さなコード片を関数に置き換えても可読性や保守性の向上には繋がらないため，関数化の対象となるコードクローンは，ある程度の大きさの処理のまとまりである必要がある．そこで，本方式では，抽出する最小深さを設定することで，指定した最小深さ以上の部分木のみを抽出し，部分木集合に格納する．これを実現するため，各ノードは，子ノードを再帰的に探索することで，葉との最大距離を算出し，保持する． また，根ノードが文となる部分木のみを抽出する．文とは，プログラムの部分構造であり，トークンを組み合わせることで処理を行う実行単位である．

3.4. 編集距離を用いた部分木間の類似度計量機能

本節では，作成した部分木の集合から各部分木間の類似度を計量する方法について述べる．構文木同士の類似度を計量する手法として，完全一致する部分木の個数から計量する手法や，部分木の編集距離から計量する手法などが挙げられる．本研究では，追加，削除，更新の操作の回数に着目して類似度を計量する手法を用いる．本手法では，編集距離の算出手法として，Zhang と Shasha によって提案されたアルゴリズム[9]を用いる．3.3 で抽出した部分木の組み合わせ全てに類似度計量を行い，部分木間の距離行列を作成する．

3.5. 式の置換による構造同一化機能

本節では，編集距離 1 以上の部分木間の構造を同一化する方法について述べる．自動関数生成は，基本的には部分木が完全一致するものについてしか行うことが出来ない．しかし，類似する部分木の一部を変形することによって同一の構造に変換することができれば，自動関数生成を適用する範囲を拡張することが可能である．ここでは，任意に定める閾値 ε 以下の編集距離を持つ完全一致ではない部分木間のうち，式に対応する部分木を置換することによって構造の同一化を実現する． 本手法は，3.4 節で編集距離の算出時に編集操作を行なったノードを対象とする．また，編集操作を行なったノードが連なっている場合，編集操作を行なったノード群のうちの最上位に位置するノードを対象とする．対象となるノードが式であり，かつ対象となるノードの兄弟ノードの個数が編集操作の前と編集操作の後で変わっていない場合にのみ本手法を適用する．前述した条件を満たす二つの部分木を，新たな同一のノードに置換する．この置換されたノードには，抽象構文木の抽象文法名として新たに expr を用意し設定する．実行例を図 2 に示す．

3.6. 自動関数生成機能

本節では，一致する部分木から関数の生成を行う方法について述べる．はじめに，ソースコードの先頭に引数，処理，戻り値を空欄とした関数部のテンプレートを挿入する．次に，対象となる二つの部分木のノードが保有する行番号を参照し，それぞれの部分木に該当するコード片を抽出する．抽出したコード片に含ま

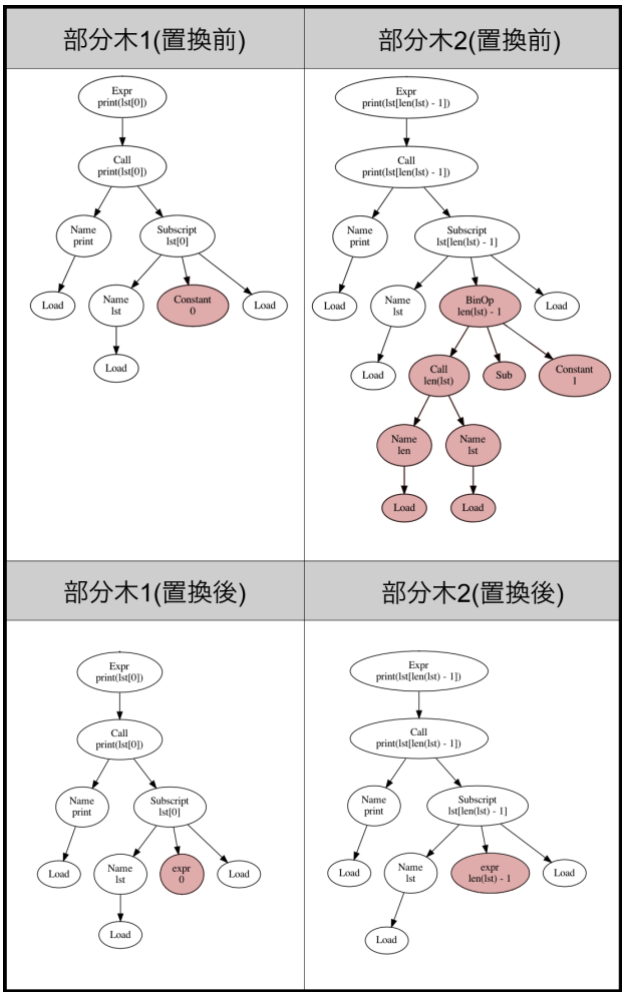

図 2 式の置換による構造同一化機能の実行例

れる式のうち，二つの部分木間で識別子が一致しない式および全ての変数を関数部で使用する変数として定義し置換する．置換したコード片をはじめに挿入したテンプレートの処理部に加える．変数として定義した式のうち，定数およびプログラム上で既に割り当て済みの識別子を引数として定義する．値の再代入やメソッドの呼び出しなどにより何らかの影響を受ける変数を戻り値として定義する．定義した引数および戻り値を，テンプレートの該当位置に加える．元のコード片をそれぞれ削除し，削除した行に，作成した関数を実行する文を挿入する．

4. 実験

本章では，同一ソースコード内に存在するコードクローン検出による自動関数生成方式の評価実験について述べる．4 章の構成について述べる．4.1 節では実験環境について述べる．4.2 節では，実験 1 として，用意した一致するコードクローンを含むソースコードに対して構文解析機能を適用し，抽出された構文木を観察

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

for _ in range(cannon1):
    if rate1 > random.random():
        cannon2_ -= 1
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    if rate2 > random.random():
        cannon1_ -= 1
    if cannon1_ == 0:
        break

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

図 3 一致するコードクローンを含むソースコード

```
# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
   for j in range(i+1, n):
      if A[j] < A[i]:
         A[i], A[j] = A[j], A[i]
print(A)

# Bubble Sort
                A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
   for j in range(n-1):
      if A[j+1] < A[j]:
         A[j], A[j+1] = A[j+1], A[j]
print(A)
```

図 4 類似するコードクローンを含むソースコード

することで，関数化における置き換えのための最適な深さについて検討する．4.3 節では，実験 2 として，提案方式である自動関数化方式を一致するコードクローンを含むソースコードに対して適用し，提案方式の有効性を検証する．その際，置換する部分木の深さパラメータを変化させて実験を行い，最適な深さについて検討を行う．4.4 節では，実験 3 として，提案方式である自動関数化方式を類似するコードクローンを含むソースコードに対して適用し，提案方式の有効性を検証する．

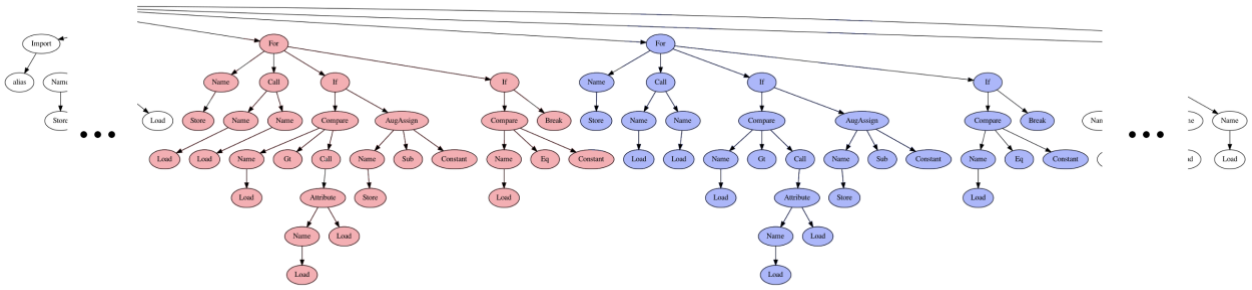

図 5 構文解析機能によって生成した抽象構文木

(関数部)

```
def function1(var1, var2):
    if var1 > random.random():
        var2 -= 1
    return var2
```

(実行部)

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

for _ in range(cannon1):
    cannon2_ = function1(rate1, cannon2_)
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    cannon1_ = function1(rate2, cannon1_)
    if cannon1_ == 0:
        break

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

(a) k = 5 の場合

(関数部)

```
def function1(var2, var3, var4):
    for var1 in range(var2):
        if var3 > random.random():
            var4 -= 1
        if var4 == 0:
            break
    return var4
```

(実行部)

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

 cannon2_ =   function1(cannon1, rate1, cannon2_)
 cannon1_ =   function1(cannon2, rate2, cannon1_)

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

(b) k = 6 の場合

図 6 実験 2 の実行結果

4.1. 実験環境

本方式は Python を用いて構築した．

実験 1 および実験 2 の対象である一致するコードクローンを含むソースコードを図 3 に示す．抽象構文木の抽出には Python の標準ライブラリである ast を用いた．実験 3 の対象である類似するコードクローンを含むソースコードを図 4 に示す．

4.2. 実験 1

4.2.1. 実験目的

実験 1 として，用意したソースコードに対して構文解析機能を適用し，抽出された構文木を観察することで，関数化における置き換えのための最適な深さについて検討する．

```
(関数部)

def function1(var1, var2, var3):
   if var1[var2] < var1[var3]:
      var1[var3],var1[var2] = var1[var2],var1[var3]
   return var1
```

```
(実行部)

# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
   for j in range(i+1, n):
      A = function1(A, j, i)
print(A)

# Bubble Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
   for j in range(n-1):
      A = function1(A, j+1, j)
print(A)
```

図 7 実験 3 の実行結果

4.2.2. 実験結果

構文解析機能によって抽出された抽象構文木を図 5 に示す．ノード数 112，最大深さ 7 の抽象構文木が生成された．本ソースコードの主な処理を表す部分木における子孫ノードの葉との最大距離 k について，ライブラリのインポートは k=1, 変数の定義は k=2, packing を使用した変数のスワップは k=3, for 文による繰り返し処理は k=6，if 文による条件分岐は k=5，print 関数による標準出力は k=3 となった．

4.2.3. 考察

本研究の目的は，コードクローンを検出し，関数化することによって，一元的に管理することで保守作業の効率を上げることである．実験 1 で得られた図 5 を確認すると，深さ 3 以下の部分木は，ソースコードにおける代入文などの 1 行の処理を表している．これらを関数化することは,保守作業の効率化に有用でない．そのため，実験 1 より，本ソースコードにおける最大深さ k は 5 以上が適切である．

4.3. 実験 2

4.3.1. 実験目的

実験 2 として，一致するコードクローンをソースコードに自動関数生成機能を適用し，提案方式の有効性を検証する．また，本研究では，コードクローン検出の対象となる部分木は，子孫ノードの葉との最大距離 k を任意の値を設定する．このとき，k の変化に対して，実行結果にどの程度違いがあるかを調査する実験を行った．

本実験では，実験 1 の結果を踏まえて最大深さ k を 5 としたときと，6 としたときの場合についてシステムを実行した．

4.3.2. 実験結果

図 3 に示した一致するコードクローンを含むソースコードに対して，k を 5 として自動関数生成機能を適用した結果を図 6(a)に，k を 6 として自動関数生成機能を適用した結果を図 6(b)に示す．k を 5 とした場合は，if 文による条件分岐文のみが関数化された．このとき，関数は二つの引数，一つの戻り値を取る．k を 6 とした場合には，先ほどの条件分岐文を含む for 文による繰り返し文が関数化された．このとき，関数は三つの引数，一つの戻り値を取る．関数化によってソースコードが簡略化できたかを考えるために，空行を除く行数を比較する．元のソースコードの行数は 15 行であった．k を 5 としたときのソースコードは，関数部 4 行，実行部 13 行の計 17 行となった．k を 6 としたときのソースコードは，関数部 7 行，実行部 7 行の計 14 行となった．

4.3.3. 考察

本実験では，k を 5 としたときと，k を 6 としたときの場合について，どちらも正常に置き換えることに成功した．また，機能のまとまりを関数に置き換えることに成功している．空行を除く行数を比較すると，k が 5 のときは行数の総計は増加したが，k が 6 のときは行数の総数を減少させることが出来た．これらのことから，適切な k の値を設定することができれば，ソースコードの保守性を向上させることが可能になると考えられる．

4.4. 実験 3

4.4.1. 実験目的

実験 3 として，類似するコードクローンをソースコードに自動関数生成機能を適用し，提案方式の有効性を検証する．

4.4.2. 実験結果

図 4 に示した類似するコードクローンを含むソースコードに対して，k を 5，式の置換による同一構造化の閾値 ε を 9 として自動関数生成機能を適用した結果を図 7 に示す．結果として，値の大小比較および更新部分が関数化された．このとき，関数化した部分木間の編集距離は 9 であった．編集操作を行なった箇所については図 8 に示す．また，k を 6 以上にした場合，関数化されなかった．また，元のソースコードが 16 行であったのに対して，実行部は関数部 4 行，実行部 14 行の計 18 行となり，コードの総量としては増えてしまったが，実行部の行数は減少した．

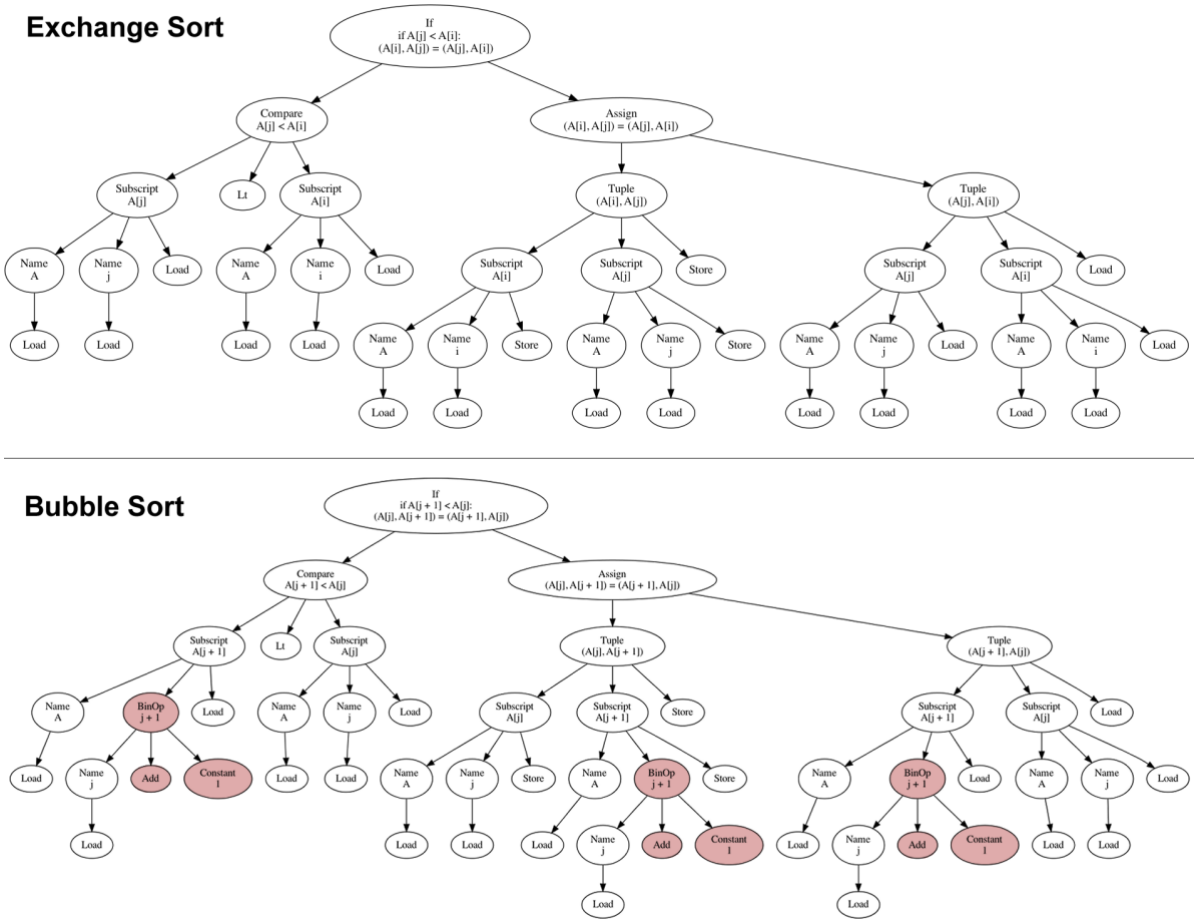

図 8 コードクローン部分の編集操作結果

4.4.3. 考察

本実験では，k を 5 としたときについて，正常に置き換えることに成功した．また，機能のまとまりを関数に置き換えることに成功している．コードの行数としては，総量としては増えてしまったが，実行部のコード量は減っており，長いソースコードに対しては関数化の恩恵が得られる可能性があると考えられる．

また，k を 6 以上としたときについて関数化されなかったのは，本ソースコード中の Exchange Sort の二つ目の for 文では range 関数の引数が二つあるのに対して，Bubble Sort では一つの引数であることが要因である．これを関数化する方法として，関数の取る引数に応じて，デフォルト引数を加えることが挙げられる．関数の取る引数の数の違いに対応することで，より汎用的に関数化を行うことができる．プログラムの記述において，一般的にネストが深いほど可読性は落ちるとされる．そのため，ネストを深める for 文を関数に置き換えることは可読性の向上に寄与すると考えられるため，for 文で頻繁に用いられる range 関数の引数の違いに対応することは，ソースコードの簡略化に寄与すると考えられる．

5. まとめ

本稿では，ソースコードの構文木表現による構造類似性を用いた自動関数生成方式について述べた．本方式では，ソースコードを構文木で表現し，構造類似性に基づいて，コードクローンを検出した．また，検出したコードクローンを生成した関数に置き換えることで，ソースコードの構造の簡略化を実現した．

また，本方式を検証するための実験システムを構築し，本方式の有効性を検証する実験を行った．

ソースコードの構文木表現による構造類似性を用いた自動関数生成方式を実現したことによって，同一機能をもつコードクローンを一元管理することが可能になり，保守作業を効率化させることができると考えられる．

今後の課題としては，自動関数生成機能を適用する深さパラメータの自動調整機能の実現と，適切な関数および変数の命名機能の実現，本方式の適用前と適用後におけるソースコードの等価性検証が挙げられる．

参 考 文 献

[1] GNU grep, http://www.gnu.org/software/grep/.

[2] 神谷年洋, 肥後芳樹, 吉田則裕, “コードクローン検出技術展開”, コンピュータソフトウェア, 28(3), pp.28‐42, 2011.

[3] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, 28(7), pp. 654-670, 2002.

[4] 吉田則裕, 服部剛之, 早瀬康裕, 井上克郎, "類義語の特定に基づく類似コード片検索法", 情報処理学会論文誌, 50(5), pp.1506-1519, 2009.

[5] M. Gabel, L. Jiang, Z. Su, "Scalable detection of semantic clones", In Proceedings of the 30th International Conference on Software Engineering, pp. 321-330, 2008.

[6] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", In 29th International Conference on Software Engineering, pp. 96-105, 2007.

[7] 石原知也, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二, "大規模ソフトウェア群に対するメソッド単位のコードクローン検出", 電子情報通信学会技術研究報告, 111(481), pp. 31-36, 2012.

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, "Clone detection using abstract syntax trees", In Proceedings of International Conference on Software Maintenance, pp. 368-377, 1998.

[9] Kaizhong zhang, Dennis Shasha, "SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS", In SIAM Journal of computing, 18(6), pp. 1245-1262, 1989.