構文的制約を利用した類似コード断片の検索方法に関する研究

2008MI226 清水 優　　2008MI236 鈴木 貴昭
指導教員　吉田 敦

1　はじめに

コードサーチエンジンは，GoogleCodeSearch[2] や Koders[1] などがあり，プログラム作成者に幅広く利用されている．検索結果から類似したソースコードを参照することで，プログラム作成者のプログラム知識向上や未完成箇所の補完ができる．プログラム作成者は，Web ページ検索と同じように目的のソースコードが含んでいると想定されるキーワードと対象言語や正規表現などの条件を指定してクエリを作成する．ゆえに，プログラム作成者はソースコードの理解と，コードサーチエンジンの条件の指定方法などを理解した上でクエリを作成する必要がある．

コードサーチエンジンは，クエリに適合した Web 上のソースコードを提示するが，クエリの表現力が弱く，複雑な条件が指定できない．ゆえに，検索結果には目的と合わないソースコードも多く含まれる．また，プログラム作成者は，キーワードを抽象化して検索範囲を広げるために正規表現に変換しながらクエリを作成し検索を行う．正規表現に変換する際, 抽象化の度合いによって, 検索結果数が変化する．抽象化の度合いが大きいと無関係なソースコードが含まれる．また，コードサーチエンジンは，クエリに適合した箇所とリポジトリサイトのアドレスを検索結果として出力する．ゆえに，プログラム作成者はソースコード毎にリポジトリサイトからソースコードを取得しなければならない．プログラム作成者はキーワードを正規表現に変換しながらクエリを作成し，検索の後，膨大な検索結果から目視で目的のソースコードを探し，ソースコード毎にリポジトリサイトにアクセスする必要があり，多くの時間を要する．

本研究では，コードサーチエンジンでソースコードの絞り込み方法を提案する．コードサーチエンジンのクエリの記述能力に限界があり，クエリだけでは類似コードが検索できない．そこで，本研究では 2 段階でソースコードの絞り込みを行う．第 1 段階にキーワードを適度な抽象化をしながらクエリを作成し，検索結果からソースコードを取得する．第 2 段階で取得したソースコードから類似コードを絞り込む検索を行う．第 1 段階は，プログラム作成者が目的のソースコードを明確に想定している状況で，その構造を具体的に記述したソースコードの断片に構文的な制約を与えて記述し，抽出した箇所を正規表現に変換してクエリを作成する．構造を具体的に記述したソースコードを入力コード断片とする．構文的な制約は，プログラム作成者毎に差異が少ない制御構造のアルゴリズムに着目することである．第 2 段階は取得したソースコードに対して入力コード断片全体をプログラム作成者が意図した箇所として絞り込みを行う．本研究では，C 言語の検索を対象とし，利用するコードサーチエンジンは GoogleCodeSearch とする．

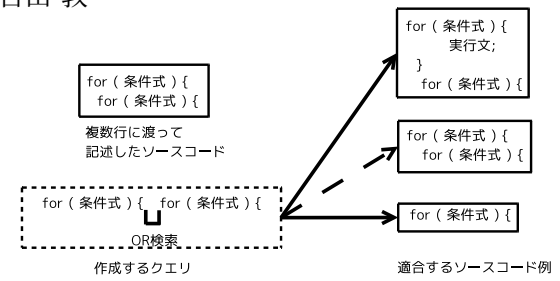

図 1　クエリと適合するソースコード例

2　GoogleCodeSearch のクエリの問題

GoogleCodeSearch はクエリに対する処理の情報が十分に公開されておらず，調査を行って推測している．クエリに関する問題として以下が挙げられる．

- 表現力が弱い．
- 記述量が多い場合，検索結果が出力されない．

GoogleCodeSearch は 1 行ごとでマッチングするので，改行を表現できない．ゆえに，プログラム作成者は複数のキーワードをクエリとして作成する際に，スペースを挿入する．GoogleCodeSearch の検索は，基本的にスペースを挿入してクエリを追加していくことで適合するファイル数は減少する．しかし，クエリによっては適合するファイルが増える場合もある．ゆえに，スペースの区切りを OR と解釈している．

1 行ごとのマッチングは複数行に渡って記述された入れ子関係のソースコードを表現できない．for の二重ループのクエリで適合するファイルの例を図 1 に示す．ソースコードからクエリを作成して検索を行うと，無関係なソースコードが含まれる．ゆえに，コードサーチエンジンは複数行に渡って記述されたソースコードの行の順序や連続した行の記述を指定できず，構文的な条件を考慮したクエリを作成できない．また，コードサーチエンジンのクエリは記述量にある程度制限があり，記述量が多いと検索結果が出力されない．

3　類似コード検索方法

本研究では，類似コードを探すために 2 段階の検索方法を提案する．第 1 段階では GoogleCodeSearch で類似している可能性があるソースコードを取得し，第 2 段階で取得したソースコードが類似しているか判別し，一致したソースコードと一致した箇所を出力する．ソー

スコードの処理の中でも，計算式はプログラム作成者によって様々な表現がありパターンとして絞り込み難いが，制御文はアルゴリズムに依存しており，プログラム作成者ごとの記述方法の差異が少ない．ゆえに本研究では制御文に着目し，文と文の間の包含関係を構文的制約とする．

3.1 プログラム作成者の意図したい箇所の指定

コード断片でプログラム作成者の意図したい箇所はプログラム作成者に依存しており，クエリを作成する際に機械的に抽出することは難しい．変数名やマクロはプログラム作成者に依存するので，ソースコードを探す際に含めることは適切でない．しかし，変数名やマクロの中でも temp や NULL はプログラム作成者で記述方法が統一的であり，プログラム作成者が意図的にクエリに入れたい場合がある．そこで，プログラム作成者が，あらかじめコード断片に意図する箇所を指定する．

指定方法はプログラム作成者が意図的に省略したい箇所や考えがうまくまとまっていない箇所は"_"(アンダースコア) で記述する．指定箇所は，関数名と変数名，引数名，型名とする．

3.2 検索手順

ツールの第 1 段階と第 2 段階の類似コードの検索手順を以下に示す．

第 1 段階

1. 構文解析したコード断片から特徴的箇所を抽出する．
2. 抽出した箇所を正規表現の変換ルールを用いてクエリを作成する．
3. GoogleCodeSearch に作成したクエリを入力し，クエリと適合したソースコードのリポジトリサイトのアドレスを取得する．
4. リポジトリサイトからソースコードを取得する．

第 2 段階

1. 構文解析したコード断片からパターンを生成する．
2. 第 1 段階で取得したソースコードを 1. で生成したパターンが一致しているか判別する．
3. 一致したソースコードと一致した箇所をプログラム作成者に提示する．

図 2 にツールで行う処理の工程を示す．コード断片を構文解析した後，第 1 段階の検索の処理工程と第 2 段階の検索の処理工程が分かれる．

4 コードサーチエンジンによる検索

4.1 特徴的箇所の分類

クエリを作成する際にコード断片から行ごとに抽出する．抽出する箇所の基準は以下とする．

- プログラム作成者に記述方法が依存しない．
- コード断片で捉える必要がある箇所．

これらの 2 点を考慮して，特徴的箇所として抽出する対象を以下に示す．

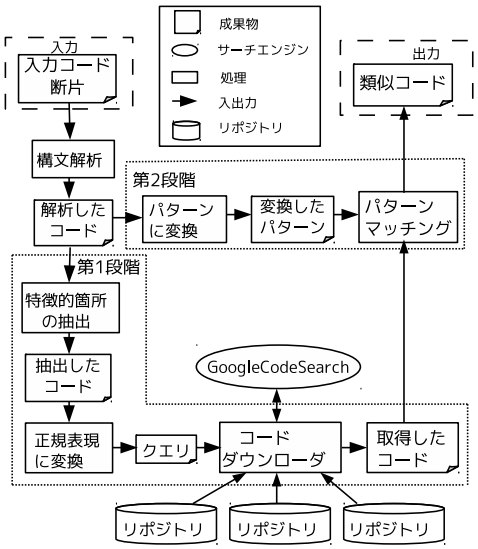

図 2　ツールの全体像

- 関数のインタフェース
- 標準ライブラリ関数に定義されている型
- 型修飾子
- 記憶域クラス指定子
- 制御文
- 一部のライブラリ関数

制御文の包含関係を表現する際に記述する中括弧は字下げスタイルの違いにより異なるので，抽出はしない．宣言文の型指定子は，int や void などの予約型，プログラム作成者が独自に定義する型や標準 C ライブラリに定義されている型であるユーザ定義型がある．標準 C ライブラリに定義されているユーザ定義型はソースコードでライブラリ関数を記述する際に宣言しなければならないことがある．プログラム作成者はある程度同じ目的で記述するので抽出をする．制御文は制御文のみの記述の場合，広く適合する．ゆえに，制御文に付随する条件式に着目し，条件式がアンダースコアで省略されていない場合抽出をする．ライブラリ関数は，標準 C ライブラリに定義されているユーザ定義型同様にプログラム作成者はある程度同じ目的で記述するので抽出をする．しかし，printf などのコード断片の特徴として適切ではないソースコードもあるので，厳選して抽出をする．

4.2 正規表現に変換する箇所

GoogleCodeSearch で検索を行う際に，ソースコードの適合範囲を広げるために必要な箇所を正規表現に変換や挿入を行う．変換箇所や挿入箇所は以下が挙げられる．

- アンダースコア
- 空白箇所
- エスケープする文字

アンダースコアは記述する箇所が関数名や型名と変数名や引数名でそれぞれ対応したメタ文字に変換する．関数名や型名は含まれる文字が英数字やアンダースコアであるので，英数字やアンダスコアに一致する\w+ に変換する．また，変数名や引数名はライブラリ関数の引数で書式文字列を囲む""を含む場合があるので，任意の文字に一致する.+ に変換する．

空白箇所はソースコードの見栄えを良くするために任意で記述する場合や連続した識別子を分けるために必ず記述する場合がある．ゆえに，それぞれ異なった対応を行う必要があり，空白必須箇所は\s+，空白任意箇所は\s*を挿入する．

4.3 複数のクエリ作成

前述したように GoogleCodeSearch のクエリは記述量にある程度制限があるので，クエリを作成する際はより少ない記述で検索を行う必要がある．そこで，本研究ではコード断片から複数のクエリを作成し，一つのクエリの記述量を減らす．複数のクエリで検索を行い，それぞれのクエリに適合したファイルを 1 つにまとめる．クエリの種類は以下が挙げられる．

- 関数のインタフェース
- 関数名
- 宣言文，実行文

関数名は，一般的にその関数がどういう働きをするか示すように命名する．例えば，ファイル一覧を取得する関数の場合，GetFileList と命名する．関数名は関数の処理全体を表現でき，関数名のみのクエリで類似コードを探せる．ゆえに，他の構文と組み合わせてクエリを作成する必要がない．また，それに加えて仮引数は関数名と組み合わせることで類似コードを探せる可能性があるので関数のインタフェースのクエリとして作成する．

実行文，宣言文は抽出した構文要素単体のクエリで検索を行うと膨大な検索結果になる．ゆえに，検索結果をある程度減らすために構文要素は組み合わせる．図 4 に図 3 のコード断片から作成したクエリを示す．この例は，ライブラリ関数 va_arg を man コマンドで調べて，サンプルコードがもっと欲しい場合に想定される入力コード断片である．このコード断片は書式文字からなる文字列を受け入れ、その書式文字に対応する型で可変個の引数を読み込み、印字するソースコードである．

4.4 関数名の命名規則の違いの対応

関数名は構成される単語が同じ場合でもプログラム作成者によって記述方法が異なる. 記述方法の違いとしては以下が挙げられる．

- Camel 記法
- Pascal 記法
- 構成される単語がすべて小文字
- アンダースコアの区切り
- 単語の順序の入れ替え

GoogleCodeSearch による検索はクエリと字面上で一致したソースコードを探す．一致する文字は大文字と小

```
void foo ( char *_ , ...){
 va_list _ ;
 _ _ ;
 va_start ( _ ) ;
 while ( _ )
  switch ( *_++ ) {
   case 's' :
      s = va_arg ( _ , char *) ;
      ……
   case 'd' :
      d = va_arg ( _ , int ) ;
      ……
   case 'c' :
      c = (char) va_arg ( _ , int ) ;
      ……
   }
  va_end( _ );
}
```

…： 抽出箇所

図 3 コード断片例

・関数の定義
void\sfoo\s*\(\s*char\s*.*\s*,\s*\.\.\.\s*\)
・関数名
foo\s*\(.*\)
・実行文，宣言文
va_list va_start while\s*\(\s**.+\s*\)
␣ ␣
switch\s*\(\s**.+\+\+\s*\)
␣
va_arg\s*\(\s*.+\s*,\s*char\s*\s*\)
␣
va_arg\s*\(\s*.+\s*,\s*int\s*\) va_end
␣ ␣
␣ :スペース(OR検索)

図 4 クエリの作成例

文字で区別されないので，記述方法が単語の大文字と小文字の違いである上位 3 つは抽出した関数名で検索できる．しかし，アンダースコアの区切りや単語の順序の入れ替えをした関数名は抽出した関数名と字面上で一致しないので，クエリを作成する際に対応する必要がある．単語の順序の入れ替えに関しては単語の数によって作成する関数名が膨大になる．ゆえに，単語の順序の入れ替えは単語の数で異なる対応を行う．対応関係については表 1 に示す．

表 1 は，GoogleCodeSearch で調査した結果である．調査は，GoogleCodeSearch に存在する関数名の構成される単語の数，アンダースコアの区切り，単語の順序の入れ替えの 3 つに関して行った．4 つ以上の単語で構成される関数名を含むソースコードは GoogleCodeSearch のソースコードに少数しか含まれていなかったので，構成される単語が 2 つと 3 つに関して調査を行った．

表 1　アンダースコアの区切りと単語の順序の入れ替えに関する対応

構成する単語の数	アンダースコアの区切り	単語の順序の入れ替え
2 つ	○	○
3 つ	○	×
4 つ以上	○	×

アンダースコアの区切りはアンダースコアの区切りと Camel 記法や Pascal 記法の関数名と存在するファイル数にあまり違いがみられなかった．ゆえに，構成される単語の数に関わらず，アンダースコアの区切りが行われていると考えられるので，すべての関数名で対応する．

単語の順序の入れ替えは構成される単語が 2 つの場合単語の順序を入れ替えていない関数名に比べ少数であるが，ソースコードが存在した．しかし，構成される単語が 3 つに関してはすべての順序の組み合わせでソースコードが存在しなかった．存在しなかった原因として，単語の品詞の構成が決められていることが考えられる．そこで，同一の品詞を含む関数名を複数用意してもう一度検索を行った．しかし，同一の品詞を入れ替えた関数名は存在しなかった．ゆえに，単語の数が増えるにつれて順序を入れ替える記述は行われていないと考える．

5　パターンマッチによる絞り込み

類似コードを検索する際に，TEBA[3] の書き換えツール rewrite.pl を使用する．構文解析した入力コード断片全体を rewrite.pl の変換前パターンに変換する．変換後パターンは変換前パターンの前後にコメント”//BEGIN_MATCH” とコメント”//END_MATCH” を挿入するパターンである．これにより，変換前パターンにマッチした箇所をプログラム作成者に提示できる．

アンダースコアは変換前パターンにする際，パターン変数に置き換えてプログラム作成者の意図しない箇所や曖昧な箇所に対応する．なお，TEBA の解析において，アンダースコアは 3 つの識別子に分類ができる．また，パターン変数の直後が”;” の場合は”$;” へ変換する．

”//BEGIN_MATCH” と”//END_MATCH” が挿入された箇所をプログラム作成者に html のリンクを使って適合箇所をプログラム作成者に提示する．

6　評価と考察

本研究で提案した手法がコードサーチエンジンによる実際の検索で有効であるか評価を行った．本研究では 2 段階による類似コードの絞り込みを行う．これらはそれぞれ異なった提案手法であるのでそれぞれで評価を行う必要がある．

6.1　評価方法

6.1.1　コードサーチエンジンによる検索

GoogleCodeSearch に存在するソースコードから関数を選択し，選択した関数ごとでクエリを作成して GoogleCodeSearch で検索を行う．検索結果から関数を選択したファイルや同一の関数を含むファイルが含まれているか目視で確認を行う．

6.1.2　パターンマッチによる絞り込み

第 1 段階で取得したソースコードを入力コード断片から作成したパターンを利用してマッチングを行い，その結果から考察を行う．

6.2　評価結果と考察

6.2.1　コードサーチエンジンによる検索

それぞれの関数で検索を行った結果，選択したソースコードや同一の関数に適合した場合は，関数名が特徴的であるか，制御文や宣言文のクエリでは条件式が特徴的かクエリとして有効なライブラリ関数を含んでいた．適合しない場合は，抽出した箇所に含まれる条件式が単純で，ソースコードをうまく絞れなかった．適合しなかった関数の制御文と宣言文のクエリで取得したソースコードには，選択した関数と中括弧の包含関係が同じソースコードが含まれていた．このことから，コード断片の特徴的箇所を捉えて，クエリを作成できたと言える．

6.2.2　パターンマッチによる絞り込み

第 2 段階の評価は，第 1 段階で選択した関数によって取得できたソースコードが異なるので，パターンマッチの有効性は類似コードが取得できた関数のみで確認した．また，アンダースコアで意図的に省略して記述することで，パターンに適合するソースコードが増えた．ゆえに，本研究の第 2 段階の検索は有効であると言える．

7　おわりに

本研究では，コードサーチエンジンで取得したソースコードから類似コードを絞り込むことを目的として，類似コードの検索を 2 段階で提案した．第 1 段階は，入力コード断片に制約を与えるようにプログラム作成者が記述することでクエリを自動的に生成できた．第 2 段階は，意図しない箇所や曖昧な箇所をアンダースコアで記述することでプログラム作成者毎の記述の違いを吸収した．

今後の課題としては，GoogleCodeSearch が終了したので，Koders などの他のコードサーチエンジンにおいても，それぞれの正規表現に対応した上で本手法の有効性を確認する必要がある．

参考文献

[1] Black Duck Software, “*Koders*,” http://www.koders.com/, 2011.

[2] Google, “*GoogleCodeSearch*,” http://www.google.co.jp/codesearch#/, 2011.

[3] 吉田敦, 蜂巣吉成, 沢田篤史, 張漢明, 野呂昌満, “属性付き字句系列に基づくプログラム書換え支援環境の試作”，ソフトウェアエンジニアリング最前線 (ソフトウェア・エンジニアリング・シンポジウム 2010 予稿集)，pp.119-126，Aug. 2010.