社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

類似した要素を検出できるブルームフィルタを用いた高速コード片検索手法

酒井　宏樹† 　石尾　　隆†† 　井上　克郎†

† 大阪大学情報科学研究科先端科学技術研究科　〒565-0871 大阪府吹田市山田丘 1-5
† 奈良先端科学技術大学院大学　〒630-0192 奈良県生駒市高山町 8916-5
E-mail: †{h-sakai,inoue}@ist.osaka-u.ac.jp, ††ishio@is.naist.jp

あらまし　本研究では既存のソースコード検索ツール NCDSearch の大幅な実行時間の削減を図る．NCDSearch は与えられたコード片に類似したソースコードを行単位で検索していくが，本研究では検索対象のファイルが検索対象のコード片を包含している可能性がある場合にのみファイル内部の検索を行うように改良する．具体的には，n-gram モデルを用いて類似コード片に対しても包含を検出できるような包含率を導入し，ブルームフィルタを用いて高速に判定を行う．評価実験では，既存研究の評価実験と同様の実験を行い，改変前の結果と比較した．その結果，再現率と処理時間の観点から本手法の有用性を確認した．
キーワード　ブルームフィルタ，コード片検索，包含率

Efficient source code search using Bloom filter capable of detecting similar elements

Hiroki SAKAI†, Takashi ISHIO††, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
† Graduate School of Science and Technology, Nara Institute of Science and Technology
8916-5 Takayamacho, Ikoma, Nara 630-0192, Japan
E-mail: †{h-sakai,inoue}@ist.osaka-u.ac.jp, ††ishio@is.naist.jp

Abstract　In this research, we try to reduce the execution time of NCDSearch, an existing source code search tool. The original method takes as input a code fragment as a query and searches its similar fragments in target files line by line. Our new method firstly tests source code inclusion relationship so that the tool can search only files which likely include the query code fragment. To test the relationship efficiently, we introduce Bloom filter with n-gram model. In an evaluation experiment, we replicated the experiment to evaluate the original method with the new method. As a result, the usefulness of this method was confirmed from the viewpoint of the accuracy and the processing time .
Key words　Bloom filter, source code search, File containment ratio

1. ま え が き

開発者はソフトウェア開発および保守において，しばしば類似したコードを書く [1]〜[3]．そのため，ソースコード内にバグが見つかった場合，開発者はそのコードの類似コードにも同様のバグが存在していないか検査する必要がある．この問題に対する修正支援技術として，類似コード片検索技術は重要であると考えられている [4]．我々の研究グループでは，企業の様々な要望を反映した類似コード片検索ツールとして NCDSearch を開発した [5]．このツールではクエリとして与えられたコード片と類似したソースコード片を，ファイル内部を行単位で移動するスライディングウィンドウを用いて検索していく．NCDSearch は類似性の判定には正規圧縮距離を用いており，変数名の変更や並び替えの変化に強く，多言語への対応も容易という特性 [6]〜[8] を活用している．一方で，正規圧縮距離の計算には時間が必要であり，企業の開発者が日常的に繰り返し使うにはまだ改善が必要な段階である．

本研究では，この検索時間を改善するために，ファイル単位

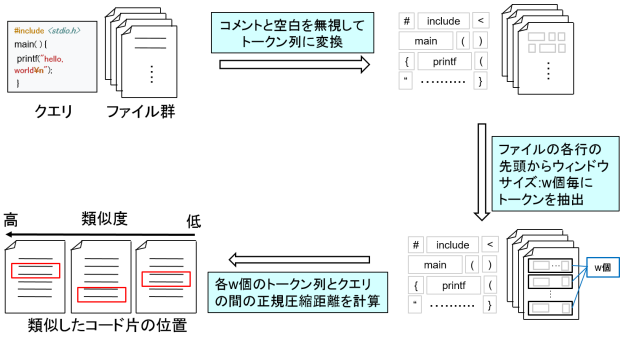

図 1　NCDSearch の処理概要

でのフィルタリングを導入する．クエリとなるソースコード片が与えられたとき，ファイル単位でそのクエリに類似したソースコード片を含んでいる可能性があるかどうかをまず判定し，含んでいる可能性があるファイルのみを行単位で検索する一方で，可能性がないファイルに対する検索を完全に省略することで実行時間を短縮する．この包含判定を，本研究では類似ソースコードにも対応する n-gram モデルと，ブルームフィルタ [9] を用いることで高速に行う．以降，2 章では研究の背景について述べる．3 章では本研究での提案手法について述べ，4 章では評価実験について述べる．5 章で妥当性への脅威について，6 章でまとめを述べる．

2. 背　　景

2.1 コード片検索ツール NCDSearch

本研究の背景として，既存ツールである NCDSearch について説明する．NCDSearch の処理概要を図 1 に示す．NCDSearch は，検索したいコード片であるクエリ q と検索対象のファイル集合 F を入力として受け取る．F に含まれる各ファイルに対して，行単位で移動するスライディングウィンドウを用いてコード片 s を抽出し，クエリとコードの正規圧縮距離 $NCD(q,s)$ を計算する．この値が閾値よりも高いとき，NCDSearch はコード片 s をクエリの類似コードとして認識し，そのファイル名，行番号，正規圧縮距離の値を報告する．

Ishio らの実験 [5] では，NCDSearch と既存のコードクローン検出ツールを比較した．3 つの OSS プロジェクトの更新履歴から類似コードのバグ修正を集めたデータセット [10] を用いて，NCSSearch はすべての類似コードを検出することができ，他のツールと比べて最も高い再現率を示した．一定以上の類似度を持つソースコード片をすべて報告するという特性上，適合率は NCDSearch が最も低い結果となったが，正規圧縮距離の値で並べ替えると報告されたコードの中で上位 20 位以内にすべての正解のコードが含まれており，既存のツールと比べても NCDSearch は有用であると結論付けている．

しかし，評価実験では 2 億行に対して 15 時間もの処理時間を要しており，対象ソフトウェアのコードの大きさによっては，日常的に繰り返し実行するツールとしては利用が難しい場合もある．この実行時間の大部分は，類似コードが見つからない多くのファイルに対しても各行に対して逐次的に時間的コストの大きい距離計算を行うために費やされている．

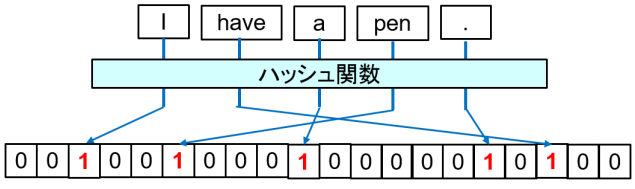

図 2　検索対象集合に対応するブルームフィルタの例

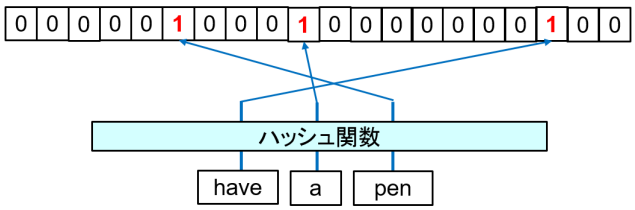

図 3　検索したい要素に対応するブルームフィルタの例

2.2 既存のコード片検索に対する高速化手法

セキュリティパッチが適用されていないファイルを高速に検出する手法として ReDeBug[11]，CLORIFI [12] が提案されている．これらの手法は，パッチ適用対象となるソースコードが持つべきトークン列をブルームフィルタ [9] で表現し，各ソースファイルがそのトークン列を完全に包含しているかを判定する．ブルームフィルタは包含関係のテストに使用される空間効率の良いデータ構造であり，長さが固定であるとみなせば時間計算量 O(1) で包含判定を行うことができるため，評価実験では 21 億行に対して 8 分の速さで該当コードを検出することに成功している．

2.2.1 ブルームフィルタの処理概要

検索したい要素を集合 X,，検索対象を集合 S とすると，X，S に対してそれぞれ m ビットのビット配列を用意する．各ビット列は最初はすべて 0 に設定されているものとする．また，k 個のハッシュ関数を用意する．これらのビット列，ハッシュ関数を用いて，以下の 3 つのステップで包含関係のテストが行われる．具体例として，以下の設定を用いる．

例で示すブルームフィルタの条件:
ビットの配列: $m = 20$
ハッシュ関数: $k = 1$
検索したい要素: {have, a,pen}
検索対象の集合: {I, have, a, pen, “.”}

STEP1；　検索対象の集合の各要素 s をそれぞれ k 個のハッシュ関数に入力し，各ハッシュ関数から得られた結果 $h(s) = i$ で得られた i 番目のビットを 1 に設定する．登録結果の例を図 2 に示す．

STEP2：　検索したい各要素 x に対しても同様にハッシュ関数を適用し，対応するビット列を構築する．例得られるビット列を図 3 に示す．

STEP3：　検索対象の集合 S および検索したい要素の集合 X

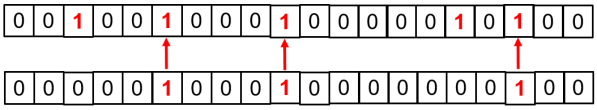

図 4　ブルームフィルタの比較

の 2 つのブルームフィルタの AND 演算を行う．この結果が検索したい要素の集合 X に対するブルームフィルタと同一である，すなわち検索対象の集合が検索したい要素の集合を完全に含んでいるとき，S が X を包含していると判断する．図 4 の例では，AND 演算の結果は X に対するブルームフィルタと同一であり，包含されていると判定される．

ブルームフィルフィルタには偽陽性がある．つまり X が S に含まれないときに包含関係であると判断してしまうことがある．これはハッシュ関数の衝突のために，つまり異なる要素が同一のビットを 1 にしてしまうために発生するもので，ブルームフィルタの偽陽性率はビット配列のサイズ m, ハッシュ関数の数 k，そして S，X の要素の数によって決まる．偽陽性の確率は，パラメータの適切な選択によって無視することができる [13].

3. 提 案 手 法

本研究では，既存研究 ReDeBug[11]，CLORIFI [12] が用いていたブルームフィルタの利用アイディアを NCDSearch に導入し，類似コードを含まないファイルを検索対象から除外することで処理速度の高速化を図る．しかし，ブルームフィルタを単純にソースコードのトークン単位で検索すると，完全一致に近いソースコードしか検出できない．そのため，NCDSearch にそのまま実装すると，クエリに全く関係のないファイルだけでなく類似コードを含むファイルも情報距離の検索対象から取り除いてしまう可能性がある．そこで本研究では，ブルームフィルタに対して包含率を定義し，部分的にソースコードが含まれるようなファイルも検索対象に含まれるようにする．

3.1 ソースコードの n-gram モデル

本研究では，ソースコードの n-gram として，UTF-8 文字コードセットにおけるバイト列上での n-gram を使用する．具体的な変換の手順を図 5 に示す．

（1） ソースコードに対して字句解析を適用し，空白やコメントを取り除いたトークン列を抽出する．C 言語におけるプリプロセッサ指令は，展開せず，そのままトークンとして扱う．

（2） 各トークンを空白（NUL 文字）で連結したバイト列に変換する．

（3） 最初のバイト列から n バイトを先頭の n-gram として取り出す．

（4） 以降，先頭位置を 1 バイトずつずらしながら，バイト列を n 個毎に新たなバイト列を生成する．ソースコードの末尾 n バイトが最後の n-gram となる．

なお，n がソースコード全体より長いとき，たとえばファイルがコメントのみからなり有効なトークンを持っていないときは，そのソースコードに対する n-gram 集合表現は空集合であるとみなす．

3.2 包含率を用いたブルームフィルタ

本研究で用いる包含率は以下のように定義する．クエリとして与えられたソースコード片に含まれる n-gram の集合 q のうち，検索対象となるファイルに含まれている n-gram の集合 f に含まれるような要素の割合である．

$$CONTAITMENT(q, f) = \frac{|q \cap f|}{|q|}$$

この包含率が閾値 θ を超えたとき，ファイル f がクエリ q の内容を含んでいる，つまりファイル f を検索すると q の内容が見つかる可能性があると判断する．n-gram モデルを用いると，ファイル内部での厳密な並び順までは考慮していないので，類似した単語を複数含むようなファイルが偶然マッチする可能性は残るが，まったく内容が無関係なファイルはこの閾値によって除外することができる．

3.3 包含率の推定

本研究では，包含率をブルームフィルタの AND 演算の結果から求める．これは単純に AND の結果で得られたビット列の 1 になっているビットを数えればよいように見えるが，ハッシュの衝突可能性があるため，直接数値を計算することはできない．そこで本研究では，ブルームフィルタの 2 つのビット列から共通した要素の数 S を推定する方法として，Odysseas らが提案している以下の式を用いる [14].

$$S = \frac{\log\left(m - \frac{t_\wedge \times m - t_1 \times t_2}{m - t_1 - t_2 + t_\wedge} - log(m)\right)}{k \times \log\left(1 - \frac{1}{m}\right)} \quad (1)$$

この式では，t_1，t_2 がそれぞれ検索したい集合，検索対象の集合に対するブルームフィルタの中で 1 に設定されているビットの数，$t_\wedge$ が 2 つのブルームフィルタで共通して 1 に設定されているビットの数，m をがブルームフィルタの長さ，k がハッシュ関数の数である．

3.4 NCDSearch に対しての実装

本研究ではハッシュ関数に SHA-1 を用いる．k 個目のハッシュ関数は，与えらえた 1 つの n-gram に対して，SHA-1 を k 回適用する（SHA-1 の値をさらにハッシュに通す）操作とする．この関数を使って，以下の手順でブルームフィルタを構築する．

- n-gram として切り出したバイト列を k 個のハッシュ関数に入力する．
- ハッシュ関数から得られた値の先頭 64 ビットを long 値として解釈し，その値をブルームフィルタの長さ m で割った余りを計算し i とする．
- ブルームフィルタ内部の i が示すビットを 1 に設定する．

ブルームフィルタを導入した NCDSearch の処理の流れを図 6 に示す．クエリ q および検索対象の各ファイル f をそれぞれブルームフィルタへと変換し，その AND を取ることでクエリ計算を行い，得られたビット列を式 (1) に当てはめることで包含率を求める．NCDSearch は Java で実装されており，ビットが 1 になっている要素の数は標準ライブラリの BitSet を用いて高速に求めている．包含率 $CONTAINMENT(q, f)$ が閾

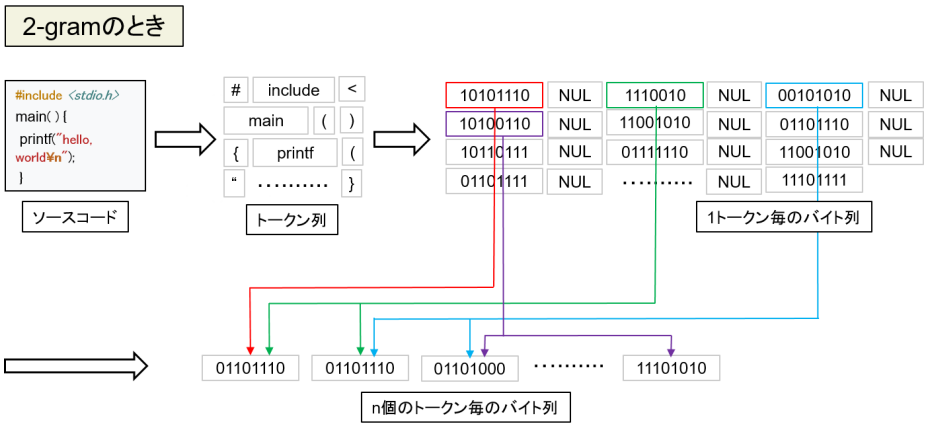

図 5　n-gram の実現方法

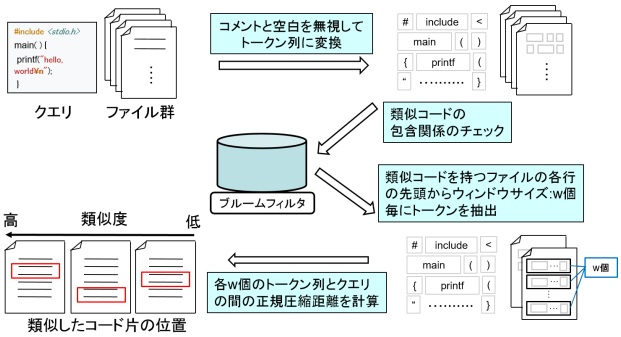

図 6　ブルームフィルタを用いた NCDSearch の処理概要

値を超えるようなファイル f に対してのみ，通常の検索，すなわちスライディングウィンドウを用いてのコード断片の抽出とクエリとの比較を実行する．

4. 評 価 実 験

本研究では，提案手法の有用性を考察するため，評価実験を行った．評価する項目として，以下の 3 つのリサーチクエスチョンを設定した．

RQ1.　改変前に比べて NCDSreach の実行時間は削減されるのか．提案手法は，ブルームフィルタを実装することで, 実行時間の削減を目指している. 改変前の NCDSearch と比較して提案手法が実行時間の削減に貢献しているかを調査する．

RQ2.　ブルームフィルタの実装によって再現率を下げることはないか．提案手法は，検索したいコードを含む可能性が低いものをファイルを情報距離の計算から除外している．そのため誤って類似コードを含む可能性のあるファイルを除外してしまう可能性もある．改変前の NCDSearch と比較して提案手法によって再現率が下がっていないかを調査する．

RQ3.　実行時間，再現率を考慮した最適なパラメータはどんな値か．提案手法で用いたブルームフィルタには，用意するビット列の長さ m，ハッシュ関数の数 k，n-gram の大きさ n，包含率の閾値 θ の 4 つのパラメータが存在することから，これらの理想的な値を調査する．

これらのリサーチクエスチョンを考察するため，本研究では，

表 1　実験対象のデータセット

Projects	#Queries	#Bugs	ファイル数 †	LOC†
PostgreSQL	14	34	1,058	277,959
Git	5	8	261	67,028
Linux	34	39	22,181	6,931,715
Total	53	81	792,432	241,074,652

† ファイル数，行数はクエリごとの中央値である．

改変した NCDSearch に対して改変前の NCDSearch に対して行った実験を同じ条件で行った．本章では，実験の設計とその結果について述べる．

4.1　実験の設計

データセットは既存研究 [10] で使用されたものを用いる．これは PostgreSQL，Git，Linux という 3 つの OSS プロジェクトにおいて，「同じ修正を複数場所に施した」と版管理システムのコミットメッセージに記載されているもののうち，類似コードが原因であると考えられるものを抽出したものである．全部で 53 件のバグ修正事例があり，各事例はクエリ，その類似コードのソースコード位置の集合からなる．各プロジェクトはいずれも C/C ++ を用いている．このデータセットの大きさなどの特性は以下の通りである．

- ほとんどのクエリには数行のコードが含まれており，その中央値は 2 である．最長のクエリは 14 行のコードである．
- 53 個クエリのうち 42 個がソースコード内に 1 つだけの類似コードを持つ．残りのクエリは 2 個以上，最大で 13 個の類似コードを持つ．
- 25 個のクエリについては，検索対象となっているコードと完全一致する内容である．

表 1 は分析されたバージョンのプロジェクトにおける各プロジェクトのクエリ数，バグ数，ファイル数，ソースコード行数を示している．クエリごとにそれぞれソフトウェアのバージョンは異なるため，ファイル数とソースコード行数は中央値を示している．

4.2　NCDSearch の設定

データセットに対して改変前後の NCDSerach を使用することで，再現率 (Recall) と実行時間を調査する．今回の評価実

表 2　提案手法のパラメータ

ビットの長さ	m
ハッシュ関数の数	k
n-gram の値	n
包含率の閾値	θ

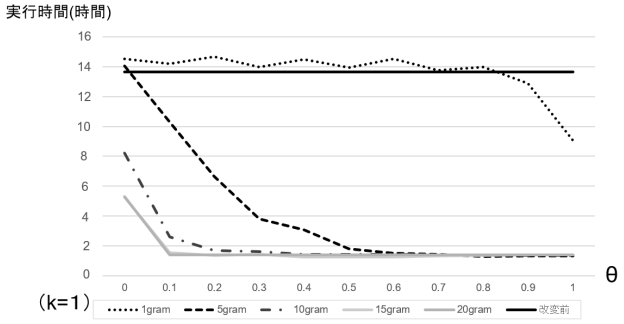

図 7　RQ1. 実行時間の削減効果

験は改変前との比較を目的とするため，改変前の評価実験 [5] と同じ条件である情報距離の閾値は 0.5 と設定する．また圧縮アルゴリズムには改変前の評価実験で最も高い再現率を示した Deflate を使用する．

表 2 に本研究の提案に由来するパラメータを示す．ブルームフィルタで使用するビット数 m は大きければ大きいほど共通要素の推測の精度が上がるため，十分に大きい数である，検索対象の集合の要素数の 1000 倍の値に設定している [14]．

4.3　RQ1. 実行時間の削減効果

ハッシュ関数の個数は $k = 1$ で固定し，パラメータとして n を 1, 5, 10, 15, 20，包含率の閾値 θ を 0.1 ずつ 0.0 から 1.0 まで動かして，ツールの実行時間を測定した．この結果を図 7 に示す．

この結果を見ると，パラメータにはよるが最大で 9 割もの実行時間を削減できた．1-gram のときは改変前よりも実行時間が大きくなっているが，5-gram 以上の n-gram では θ が 0.5 以上のものはすべて 9 割の実行時間を削減している．これは 1-gram だと文字列の順序が関係なく, クエリの要素が順序関係なくファイルにすべて存在すればクエリがファイルに存在すると判断してしまうので，ブルームフィルタの計算に加えてほとんどのファイルに対しても情報距離の計算を行ったためであると考えられる．n-gram が大きくなるほど閾値にかかわらず実行時間が小さくなっているのは，ファイルのフィルタリングが機能した効果であると考えられる．

4.3.1　RQ2. 再現率への影響

RQ1 の実験と同じく，ビット数は十分に大きい和として検索対象の集合の要素数の 1000 倍に設定し，$k = 1$，$n = 1, 5, 10, 15, 20$，θ を 0.1 ずつ 0.0 から 1.0 まで動かして実験を行った．図 8 が再現率の結果である．この結果を見ると，パラメータにもよるが，改変前と同じように再現率として 1 を示している部分がある．また，n-gram が大きくなるほど閾値にかかわらず再現率は低下する．これは n-gram が大きくなるほど，クエリと集合の要素が一致しにくくなり，類似コードに対する包含率が低下したためであると考えられる．ただし，20-gram まで，再現率が 1 となるパラメータは存在している．

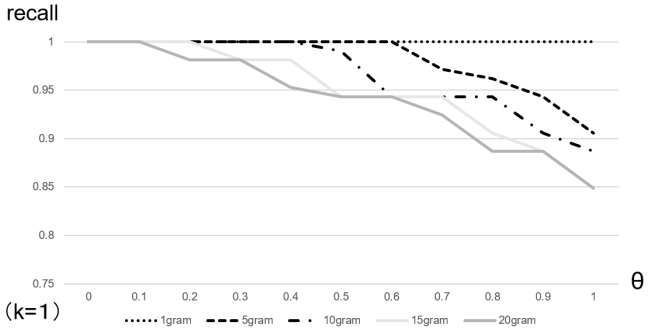

図 8　RQ2. 再現率への影響

表 3　理想のパラメータ

5-gram	0.5-0.6
10-gram	0.2-0.4
15-gram	0.1-0.2
20-gram	0.1

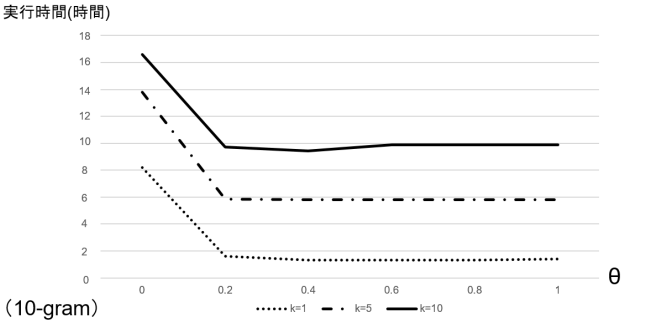

図 9　RQ3. ハッシュ関数に対しての実験結果（実行時間）

表 4　RQ3. ハッシュ関数に対しての実験結果（再現率）

	θ					
	0.0	0.2	0.4	0.6	0.8	1.0
K=1	1.000	1.000	1.000	0.943	0.943	0.887
K=5	1.000	1.000	1.000	0.943	0.925	0.887
K=10	1.000	1.000	1.000	0.943	0.925	0.887

4.3.2　RQ3. パラメータの評価

RQ1，RQ2 の実験結果から，9 割の実行時間を削減でき，同時に再現率が 1 を保つようなパラメータが存在した．そのパラメータを表 3 に示す．この表を見ると，10-gram が最もパラメータの範囲が広い．これは 10-gram が最もクエリに対する，類似コードを含んだファイルとそうでないファイルの包含率の差が大きくなると言い換えられる．そのため $n = 10$ として，包含率の閾値を範囲の中間の値である $\theta = 0.3$ とするときが安定的に再現率を 1 に保ちつつ，処理時間を最大限に削減できるパラメータではないかと考えられる．

次に理想的な n-gram である 10-gram に対して，ハッシュ関数の数 k，包含率の閾値 θ をパラメータとして実験を行った．

その結果を図 9，表 4 に示す．この結果を見ると，ハッシュ関数の数に比例して実行時間が大きくなっていることがわかる．これはハッシュ関数が増える分，ハッシュ計算が増えるためである．また，再現率に関してはハッシュ関数によって値がほとんど変わっておらず，むしろ増えると下がる部分も存在する．そのため，ハッシュ関数は少ないほど実行時間が少なく，再現率も高くなるので，ハッシュ関数の数は 1 つが理想的である．

5. 妥当性への脅威

本研究の評価実験では 3 つのオープンソースから構築したデータセットを使用したが，このデータが一般的なバグ，修正の特徴に準じていると仮定している．本手法の有用性および最適なパラメータの検証については，他のデータセットで実験すると本研究の結果と同様の結果が得られない可能性がある．アルゴリズム自体は決定性の振る舞いであるが，トークンの表現に使う文字コードやハッシュ関数などにも影響を受けている可能性がある．

6. ま と め

本研究では，ブルームフィルタを用いることで NCDSearch の実行時間の短縮を図る手法を提案した．提案手法では n-gram を用いて完全一致のコードだけでなく類似コードの包含の可能性を考慮した包含率を定義し，ブルームフィルタによって高速に集合の比較を行った結果から，包含率を推定する．既存の NCDSearch に対して行われた評価実験を提案手法に対しても同様に実施し，NCDSearch の再現率を低下させずに約 9 割の実行時間の削減をすることができた．そして，安定的にその結果を出すことのできるパラメータの推測ができた．

今後の課題としては，データセットの拡張によるパラメータの安定性の評価や，検索対象のソースコードの特性に合わせた自動的なパラメータの調整が挙げられる．

謝　　辞

本研究は JSPS 科研費 JP15H02683，JP18H03221 の助成を得た．

文　　献

[1] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, Vol. 35, No. 5, pp. 73–88, October 2001.

[2] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 447–456, New York, NY, USA, 2010. ACM.

[3] Ruru Yue, Na Meng, and Qianxiang Wang. A characterization study of repeated bug fixes. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pp. 422–432. IEEE, 2017.

[4] Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. Transferring code-clone detection and analysis to practice. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pp. 53–62, Piscataway, NJ, USA, 2017. IEEE Press.

[5] Takashi Ishio, Naoto Maeda, Kensuke Shibuya, and Katsuro Inoue. Cloned buggy code detection in practice using normalized compression distance. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 591–594. IEEE, 2018.

[6] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, Vol. 50, No. 7, pp. 1545–1551, 2004.

[7] Liang Zhang, Yue-ting Zhuang, and Zhen-ming Yuan. A program plagiarism detection model based on information distance and clustering. In *Intelligent Pervasive Computing, 2007. IPC. The 2007 International Conference on*, pp. 431–436. IEEE, 2007.

[8] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. Similarity of source code in the presence of pervasive modifications. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pp. 117–126. IEEE, 2016.

[9] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, Vol. 13, No. 7, pp. 422–426, 1970.

[10] Jingyue Li and Michael D. Ernst. Cbcd: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 310–320, Piscataway, NJ, USA, 2012. IEEE Press.

[11] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 48–62. IEEE, 2012.

[12] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. Clorifi: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 6, pp. 1900–1917, 2016.

[13] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, Vol. 1, No. 4, pp. 485–509, 2004.

[14] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, Vol. 28, No. 2-3, pp. 119–156, 2010.