修士学位論文

題目

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

指導教員

井上 克郎 教授

報告者

酒井 宏樹

平成 31 年 2 月 6 日

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 ソフトウェア工学講座

平成 30 年度 修士学位論文

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

内容梗概

ソフトウェアの開発において，コードの再利用は頻繁に行われている．しかし, 再利用されているコードにバグを見つかれば, 他の再利用されたコードにもバグが含まれる可能性があるため, 開発者は他の再利用されているコードも調査しなければならない [1],[2],[3]. そのため, 類似コード片の検索技術は, 同じバグを複数の場所で一度に修正できるため, ソフトウェア開発において重要だと考えられている [4]. そのため, 類似コード片を検索するツールが数多く開発されている. その中の 1 つである NCDSearch[5] というツールは, 検索したいコード片（クエリ）と検索対象のソースファイル群を入力として受け取り, クエリと類似したソースコード片の出現位置を報告する. このツールは複数のプログラミング言語の検索に対応するためにソースコード比較に正規化圧縮距離を採用し，検索対象ファイルの各行に対して網羅的にクエリとの比較を行う．その結果，既存研究の評価実験での適合率は 100% であり, 既存ツールと比べて見逃しが少ない反面，検索対象の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，検索に長い時間を要するという問題点がある.

本研究では,NCDSearch の大幅な実行時間の削減を図る. あらかじめ類似したソースコード片を含むファイルだけを抽出し, それらのファイルのみに対して距離計算を行うことで実行時間の短縮が可能になる. 本手法では, ファイルの抽出に高速に包含関係をテストできるブルームフィルタ [6] を用いている. また, 類似した要素も検出できるように包含率を用いることで, 類似したソースコード片を含むファイルを抽出も可能にしている. 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, 適合率と処理時間の観点から本手法の有用性を確認できた.

主な用語

ブルームフィルタ
包含率
コード片検索

目 次

1　はじめに **3**

2　背景 **5**

2.1　既存の類似コード片検索ツール 5

2.1.1　NCDSearch の処理概要 5

2.1.2　各ツールに対する評価実験 7

2.1.3　NCDSearch の問題点 10

2.2　コード片検索における高速化手法 10

2.2.1　ブルームフィルタの処理概要 11

3　提案手法 **14**

3.1　包含率を用いたブルームフィルタ 14

3.1.1　$|q \cap f|$ の推測 14

3.2　n-gram の実現方法 15

3.3　ブルームフィルタの実現方法 15

3.4　包含率の計算 16

3.5　NCDSearch に対しての実装 16

4　評価実験 **17**

4.1　実験の設計 18

4.2　NCDSearch の設定 18

4.3　実験の考察 19

4.3.1　RQ1. 改変前に比べて NCDSreach の実行時間は削減されるのか . . . 19

4.3.2　RQ2. ブルームフィルタの実装によって適合率を下げることはないか . 19

4.3.3　RQ3. 最適なパラメータはどんな値か 23

4.3.4　ハッシュ関数 23

5　妥当性への脅威 **25**

6　まとめ **26**

謝辞 **27**

参考文献 **28**

1 はじめに

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],[2],[3]. これは開発中の他のソフトウェアの再利用や, ソフトウェアの進化の過程で発生する. そのため, 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も検査する必要がある.

そのため, 類似コード片検索技術は, 同じバグを複数の場所で修正することが容易になるため, ソフトウェア開発において重要と考えられている [4]. 類似したコード片を検索するために様々なツールが開発されている.Kamiya らは CCFinderX[7] を開発した. このツールではサフィックスツリーを用いて類似性を判別し, 類似コードを検出している. しかし, 言語の入れ替えに弱い.NiCad[8] では TXL を組み込んだクローン検出ツールであり, 文字列の類似性を使ってソースコードを比較している. Ishio らは NCDSearch[5] を開発した. このツールでは正規圧縮距離を用いて類似性を判別し, 類似コードを検出している. これは gzip や xz などの圧縮アルゴリズムによって 2 つのコードを連結して圧縮すると, コード間の差分が少ないほど圧縮がきき, 各コードをそれぞれ単体で圧縮したデータとほとんど同じサイズになる特性を利用している. このアルゴリズムは変数名の変更や並び替えの変化に強いという特徴があり, 多言語にも対応している [9],[10],[11]. この 3 つのツールで比較実験をしたところ,NCDSearch が最も高い適合率で類似コードを. 検出できることが分かった. しかし, 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, 近年ソフトウェアが大規模化しているなかで, コードの大きさによっては非実用的であることが NCDSearch の問題点として挙げられる. これは NCDSearch はクエリと全ファイルの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処理時間が長くなってしまうためであると考えられる. また, コード検索に対してブルームフィルタが用いられている. ReDeBug[12],CLORIFI では [13], セキュリティパッチが適用されていないコードを見つけるために, ソースコードとパッチ適用前のコードに対して, 時間的コストの大きい検出処理をする前にブルームフィルタ [6] を使用することで高速に該当するコードを見つけることができている. 評価実験では 21 億行に対して 8 分の速さで, 該当のコードを検出することができている. 本研究では, ブルームフィルタを用いることで NCDSearch の大幅な実行時間の削減を図る. ブルームフィルタにより, あらかじめ類似したソースコード片を含むファイルだけを抽出し, それらのファイルのみに対して距離計算を行うことで実行時間の短縮が可能になる. しかし, これらで使用されているブルームフィルタは包含関係の要素しか検出できず, 完全一致のコードしか検出できない. このブルームフィルタを NCDSearch に同じように実装すると検出したい類似コードを情報距離の検索対象から取り除いてしまう.

そこで本手法ではブルームフィルタに対して包含率を新たに用いることで, 類似した要素

も検出可能にした.

この包含率を用いたブルームフィルタを使用することで, ファイル群からクエリが含まれるファイルを抽出し, それらだけに対して距離の計算を行うことで, 類似コードを検出可能なまま実行時間の削減を可能にしている. 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,5 章で妥当性への脅威について述べる．最後に，6 章を本研究のまとめとする．

2　背景

本章では，研究の背景としてまず既存の類似コード片検索ツールに関して述べる．次に類似コード検索ツールに対して行われている高速化手法について紹介する．

2.1　既存の類似コード片検索ツール

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 [1],[2],[3]. これは開発中の他のソフトウェアの再利用や, ソフトウェアの進化の過程で発生する. そのため, 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も検査する必要がある. このため, コードクローン検索技術は, 同じバグを複数の場所で修正することが容易になるため, ソフトウェア開発において重要と考えられている [4]. 類似したコード片を検索するために様々なツールが開発されている．Kamiya らは CCFinderX[7] を開発した. このツールではサフィックスツリーを用いて類似性を判別し, 類似コードを検出している.NiCad[8] では TXL を組み込んだクローン検出ツールであり, 文字列の類似性を使ってソースコードを比較している.Deckard[14] はソースコードを AST に変換し,AST から生成された特徴ベクトルを比較して近似ツリーの類似性に基づいてクローンコードを見つけることによって類似性を計算する.iClones[15] はプログラムのいくつかのリビジョンをとおしてトークンベースの増分クローン検出する.Isio らは NCDSearch[5] を開発した. このツールでは正規圧縮距離を用いて類似性を判別し, 類似コードを検出している. これは gzip や xz などの圧縮アルゴリズムによって 2 つのコードを連結して圧縮すると, コード間の差分が少ないほど圧縮がきき, 各コードをそれぞれ単体で圧縮したデータとほとんど同じサイズになる特性を利用している. このアルゴリズムは変数名の変更や並び替えの変化に強いという特徴があり, 多言語にも対応している [9],[10],[11]. この NCDSearch は他の類似コード片検索ツールとの比較実験が行われている. その実験では入出力の関係から CCFinder,NIcad に対して比較が行われており, 検出の結果,NCDSearch が最も高い検出精度示した．以下ではこの NCDSearch について詳しく説明する．

2.1.1　NCDSearch の処理概要

NCDSearch では図 1 のように処理を行う. 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群:F を入力として与える. これらをスライディングウィンドウを用いて F からコード片を抽出し, クエリとコードを比較する. それらが互いに類似している場合, そのコード片をクエリの類似コードとして認識する. 類似の概念として以下のように, クエリ q に対し

て情報距離が閾値よりも小さくなる集合 S を抽出する．

$$S = \bigcup_{f \in F} s \in W(f,q)|NCD(q,s)th$$

W（f,q）はスライディングウィンドウ,NCD（q,s）は距離関数である．最終的には類似コードとして抽出された S 内の重複するコード片をフィルタリングし, 結果のコード類似度の大きい順に報告する．

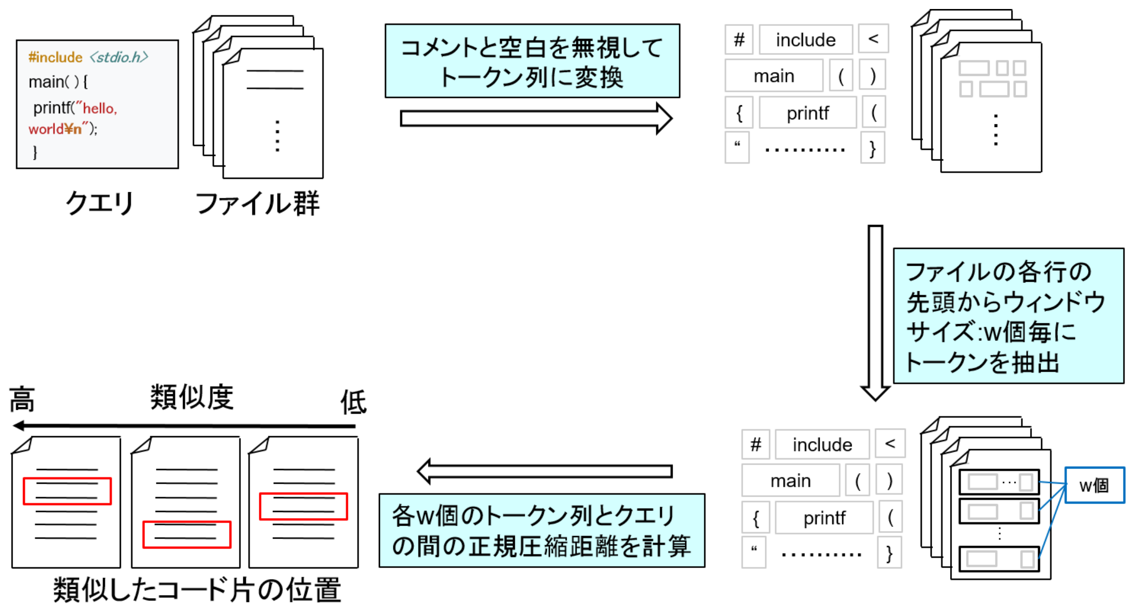

図 1: NCDSearch の処理概要

スライディングウィンドウ

サイズが可変のスライディングウィンドウ W（f,q）は, 図 2 のようにファイル内で行単位で移動し, ソースファイル f からコード片を抽出する.$|q|$ をクエリ q 内のトークンの数とすると,$0.80|q|$,$0.85|q|$, $0.90|q|$,$0.95|q|$,$|q|$,$1.05|q|$,$1.10|q|$,$1.15|q|$,$1.20|q|$ をスライディングウィンドウのサイズとしている. ここでのトークン数はコメントと空白を無視したトークンの数を表している．

コード比較

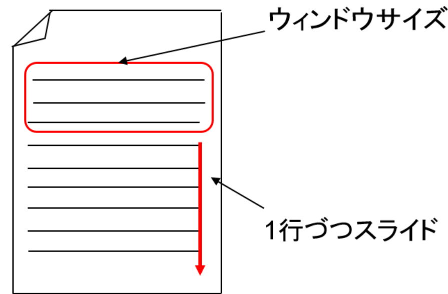

図 2: スライドウィンドウ

このツールは, 次のように定義された正規化圧縮距離を使用して 2 つのコード片の比較を行う [16].

$$NCD(q,s) = \frac{C(qs) - minC(q), C(s)}{maxC(q), C(S)}$$

データ圧縮アルゴリズムを適用するために, トークン列 q と s を連結してバイト列に変換する. 圧縮アルゴリズムには zip と gzip の最も一般的なアルゴリズムである zlib の Deflate アルゴリズムを使用している. そのため, 使用者は自分の環境でアルゴリズムのプロパティを簡単に分析できる.

フィルタリング

このツールは, 類似コード片として多数の重複するコード片を検出することがある. そのような場合は重複しているコード片の中で最も類似したコード片, つまり, クエリとの情報距離が最短であるものを類似コードとして報告します. 情報距離が最短であるコードが複数存在する場合, その中で最短のコード片を類似コードとして報告する. 報告された各コード片には, ファイル名, 最初と最後の行番号, およびクエリとの情報距離の 4 つの属性があり, 情報距離が短い順に表 1 のように表示される.

検索したいコード

$perm_fmgr_info(typeStruct->typoutput, \&(prodesc->arg_out_func[i]));$

2.1.2 各ツールに対する評価実験

ishio らはこの NCDSearch に対して既存ツールとの比較実験を行った. この実験ではツールの入出力の仕様から CCFinderX,NiCad を選択している.

表 1: NCDSearch の出力例

順位	コード位置	情報距離	検出したコード
1	pltcl.c1379-1380	0.036	$perm_fmgr_info(typeStruct- > typoutput,$ $\&(prodesc- > arg_out_func[i]));$
2	plperl.c2132-2133	0.036	$perm_fmgr_info(typeStruct- > typoutput,$ $\&(prodesc- > arg_out_func[i]));$
3	pltcl.c1329-1331	0.277	$perm_fmgr_info(typeStruct- > typinput,$ $\&(prodesc- > result_in_func));$
4	plperl.c 2088-2089	0.289	$perm_fmgr_info(typeStruct- > typinput,$ $\&(prodesc- > result_in_func));$ $prodesc- > result_typioparam =$ $getTypeIOParam(typeTup);$
5	plpython.c 1855-1856	0.361	$perm_fmgr_info(typeStruct- > typoutput,$ $\&arg- > typfunc);$ $arg- > typoid = HeapTupleGetOid(typeTup);$

評価方法

この評価実験では,PostgreSQL,Git,Linux という 3 つの OSS プロジェクトを基にした CBCD ツールのベンチマークデータセットで採用されたデータセットを用いている [17]. このデータセットは各ツールの更新履歴から抽出された完全一致コード, 類似コードを含む 53 件のクエリ, 検索結果の組の集合から成る. プロジェクトの主なプログラミング言語は C / C ++である. また, クエリには, 次のプロパティがある.

- 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, 最長のクエリには 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ち, 残りのクエリはソースコード内に最大 13 の複数の類似コードを持つ.
- 25 個のクエリが合, ソースコード内に完全一致のコードを持つ.

表 2 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, ファイルのサイズを示している.

表 2: 評価実験におけるデータセット

Projects	#Queries	#Bugs	Median #Files	Median 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

このデータセットに対して各ツールを使用することで以下の指標を扱う．適合率, 検出時間を調査した．指標としては以下を扱う．

Precision:報告された類似コードのうち正解の数
Recall:正解のうち報告された数
MAP:各ツールの平均 recall

ツール構成

CCFinder はデフォルトで少なくとも 50 個のトークンを持つコードクローンを検出する. また, デフォルトのしきい値の影響を分析するために, 長さがクエリと同じコードクローンも抽出している．NiCad では,2 つのレベルのブロックレベル, ファンクションレベルが 3 行と 10 行のコードからなる 4 つの構成を試している.NCDSearch では正規化された圧縮距離は, 圧縮アルゴリズムに依存する. その効果を評価するために,Deflate,zstd, および xz を使用する. 情報距離の閾値は 0.5 に設定している.

結果

表 3 は, 報告された類似コードの数,precision,recall, 各ツールの平均精度（MAP）をまとめたものである

.CCFinderX と NiCad にはランキング 機能がないため,MAP 値は存在しない.racall はどの圧縮アルゴリズムを用いていても NCSSerch が最も高く, すべての類似コードを検出することができているが,precision は 1 番低い結果となっている. 図は, 報告された類似コード片のランクの分布であり, 上位 20 位以内のコード片に正解の大部分が含まれていることを示している．recall が 2 番目に高いのは CCFinderX であった.recall が 1 以下になってしまうのはタイプ 3 のクローンが検出できないためだと考えられる.NiCad は他のツールに比べて最も recall が小さく, 類似コードの検出精度が悪いといえる. 報告する類似コードが少ないことから precision は高く,CCFinderX と似た値になる．こ

表 3: 評価実験の結果

Conguration	#Report	Precision	Recall	MAP
NCD (Deate, th = 0.5)	8107	0.010	1.000	0.741
NCD (zstd, th = 0.5)	368355	0.001	1.000	0.721
NCD (xz, th = 0.5)	46757797	0.001	1.000	0.722
CCFinderX (50 tokens)	70	0.629	0.728	N/A
CCFinderX ($\|q\|$ tokens)	367021	¡0.001	0.753	N/A
NiCad (Block, 3 lines)	19	0.632	0.3	N/A
NiCad (Block, 10 lines)	18	0.611	0.212	N/A
NiCad (Functions, 3 lines)	21	0.667	0.189	N/A
NiCad (Functions, 10 lines)	20	0.650	0.176	N/A

の結果から, 他のツールに比べ NCDSearch は最も高い検出 recall を示し,precision は悪いが, 類似度順に並べられた中で上位２０位以内に大部分の正解が入っている点では precision の低さを補えており, 最も有用なツールであることがわかる. また NCDSearch で使用される圧縮アルゴリズムについては, どのアルゴリズムでも recall は変わらないが,precisionMAP 共には Deate の値が最も高い. そのため, 圧縮アルゴリズムは Deate を使用するべきである.

2.1.3 NCDSearch の問題点

表 4 は, 比較実験における各ツール処理時間を示している. この処理時間は Xeon E5-2690v3 プロセッサ,64 GB DRAM, および SSD で動作する Windows 10 の一般的な商用の環境で測定されており, 各ツールには単一の制御スレッドを使用している. この表から,3 つの OSS プロジェクトの 241,074,652 行に対して NCDSearch は 12 時間 26 分もの処理時間がかかっている. 近年, ソフトウェアが大規模化しており, 自動車産業ではコードが 1 億行を超えることもあり,NCDSearch はソースコードの量によっては非実用的だといえる.NCDSearch の実行時間が大きい理由として, クエリと各ファイルのウィンドウサイズの情報距離の計算を逐次的に行っているためであるとことが考えられる.

2.2 コード片検索における高速化手法

ReDeBug[12],CLORIFI では [13], セキュリティパッチが適用されていないコードを見つけるために, ソースコードとパッチ適用前のコードに対して, 時間的コストの大きい検出処理

表 4: 各ツールのデータセットに対する処理時間

Conguration	Total Time
NCD (Deate, th = 0.5)	12h 26min
NCD (zstd, th = 0.5)	4h 35min
NCD (xz, th = 0.5)	176h 38min
Normalized LD (th = 0.5)	2h 45min
CCFinderX (50 tokens)	21h 30min
CCFinderX ($\|q\|$ tokens)	6h 22min
NiCad (Block, 3 lines)	24h 53min
NiCad (Block, 10 lines)	27h 24min
NiCad (Functions, 3 lines)	35h 53min
NiCad (Functions, 10 lines)	35h 14min

をする前にブルームフィルタ [6] を使用することで高速に該当するコードを見つけることができている. 評価実験では 21 億行に対して 8 分の速さで, 該当のコードを検出することができている. 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, 時間計算量 O(1) である. 以下ではブルームフィルタの挙動を説明する.

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. またビット列に加えて 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 3 つのステップで包含関係のテストが行われる. 実際の以下の例を用いて説明する.

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 個
検索したい要素: {have , a ,pen}
検索対象の集合: {I , have , a , pen , .}

STEP1 図 3 のように, 検索対象の集合の要素 s を全て k 個のハッシュ関数に入力する, 各ハッシュh（s）= i に対して, ビット列の i 番目のビットを 1 に設定する.

STEP2 図 4 のように, 検索したい集合の要素 x を全て k 個集合は. のハッシュ関数に入力する, 各ハッシュh（x）= i に対して, ビット列の i 番目のビットを 1 に設定する.

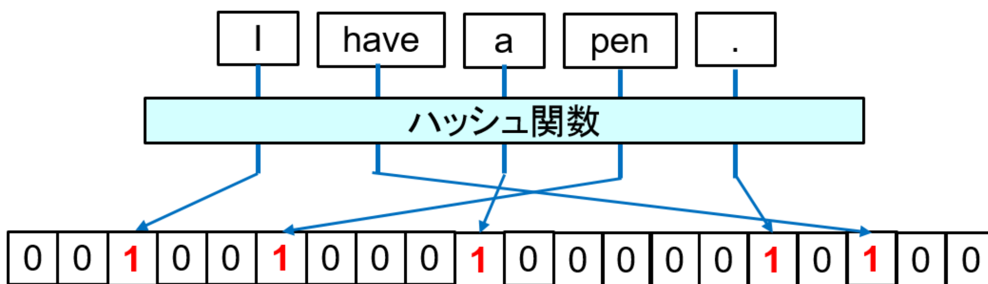

図 3: ブルームフィルタの処理 STEP1

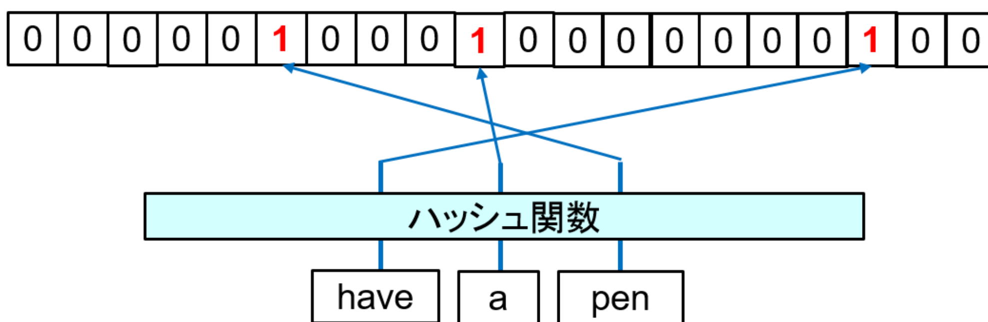

図 4: ブルームフィルタの処理 STEP2

STEP3 図 5 のように, 検索したい集合に対するビット列の中で 1 に設定されているビットの位置を同と位置を検索対象の集合のビット列参照し, すべて 1 に設定されていれば検索したい集合 X は検索対象の集合 S に含まれると判断する.1 つでも一致しなければ包含関係にないと判断する.

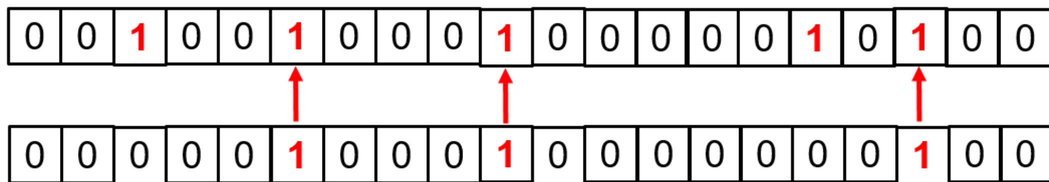

図 5: ブルームフィルタの処理 STEP3

ブルームフィルフィルタには偽陽性, 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発生する. 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）, および 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 [18].

3 提案手法

本研究では, 高速に包含関係をテストできるブルームフィルタを使用することでNCDSearchの処理速度の高速化を図る. あらかじめ, 類似コードを含まないファイルをフィルタリングすることで, 時間的コストの大きい情報距離の計算を行うファイルが少なくなり, 実行時間の削減を可能にする. しかし, ブルームフィルタは包含関係の要素しか検出できず, 完全一致の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. そこで, 本手法ではさらに, ブルームフィルタに対して包含率を定義する. 包含率を使うことで包含関係の要素しか検出できないブルームフィルタが類似した要素も検出することが可能となる.

3.1 包含率を用いたブルームフィルタ

本手法では上述のブルームフィルタに対して包含率を新たに用いる. 包含率の概念は, 以下のような, 検索したい集合の要素数に対しての, 検索対象の集合の要素と含まれる検索したい集合に共通する要素数の割合である.

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

この包含率が闘値を超えたものを類似した要素として検出することができる. 例をあげると, 検索したい集合 you,have,a,pen,., 検索対象の集合 I,have,a,pencil,. とすると, 包含率を用いないブルームフィルタを使用すると you,pen が検索対象の集合に含まれないため, 類似要素として検出することができない. しかし包含率を用いると,5 語のうちの 3 語である have,a,. が検索対象の集合に含まれるため, 包含率は 0.6 となり, 闘値の設定が 0.6 以下であれば検索したい集合を類似要素として検出することができる. しかし,$|q \cap f|$ はビット列から単純に得ることはできない.以下に $|q \cap f|$ の推測方法を述べる.

3.1.1 $|q \cap f|$ の推測

包含率の計算に必要な $|q \cap f|$ は, ハッシュの衝突する可能性があるため, 検索したい集合に対するビット列の 1 に設定されている数に対しての, 単純に二つのビット列の共に 1 に設定されているビットの数の割合で求めることはできない. そこで本手法では,Odysseas らが紹介している, ハッシュ衝突確立を考慮した, ブルームフィルタの二つのビット列から共通した要素の数を推定する方法を用いている [19]. それは,t_1,t_2 をそれぞれ検索したい集合, 検索対象の集合に対するビット列の中で 1 に設定されているビットの数,$t_\wedge$ を検索したい集合, 検

索対象の集合に対するビット列の中で共に 1 に設定されているビットの数,m をビットの長さ,k をハッシュ関数の数とすると, 以下の式で表される.

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

また, 今回の包含率には n-gram を使用する. 任意の文字列や文書を連続した n 個の文字で分割するテキスト分割方法である. 例えば上記の検索したい集合 you,have,a,pen,., 検索対象の集合 I,have,a,pencil,. を 3-gram で分割すると, それぞれ you have,have a,a pen,pen .,I have,have a,a pencil,pencil . となる.n-gram を使用することで文字列の単一の要素だけでなく, 文字列の順序関係も保存できるため, より精度の高いフィルタリングができると考えている.

3.2 n-gram の実現方法

文字列に n-gram を使用するために図 6 のような方法を用いる.

- クエリ, ファイルのソースコードから空白やコメントを取り除いたトークン列を抽出
- 各トークンをバイト列に変換
- 最初のバイト列から n 個のバイト列をまとめて新たなバイト列を生成する
- 以降同じようバイト列を 1 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

3.3 ブルームフィルタの実現方法

ブルームフィルタで必要なビット列には中身がすべて 0 である. 配列を用意する. またハッシュ関数には SHA-1 を用いている. ブルームフィルタを実現するために以下の方法を用いる.

- n 個づつにまとめて新たに生成したバイト列をハッシュ関数に入力
- ハッシュ関数から得られた値をビットの長さ m で割った余りを計算
- 余りが示すビット列の中身を 1 に設定する.
- ハッシュ関数が k 個に設定されている場合は直前にハッシュ関数から得られた値を STEP2 から STEP3 の処理を k-1 回行うことで実現する.

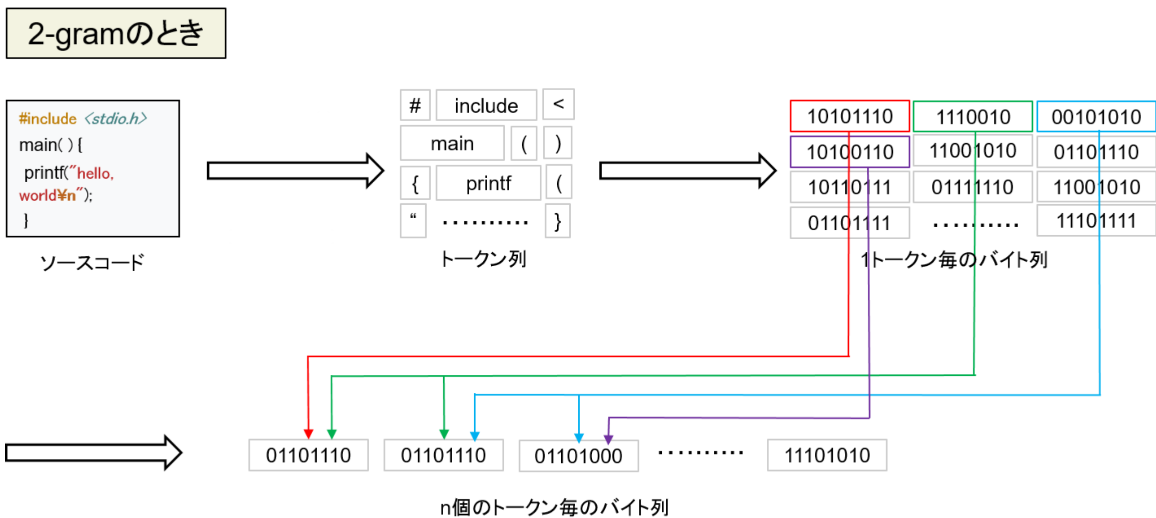

図 6: n-gram の実現方法

3.4 包含率の計算

包含率の計算を行うためには, 検索対象の集合に対するビット列の中で共に 1 に設定されているビットの数を計算する必要がある. これ求めるために,2 つのビット列の中身を確認し, 比較することは時間的コストが大きい. そこで 2 つのビット列に対して AND 演算を用いる.AND 演算を用いることで高速に二つのビット列の共に 1 に設定されているビット数を求めることができる.

3.5 NCDSearch に対しての実装

包含率を用いたブルームフィルタを図 7 のように実装する. まずクエリ:q とファイル:F 入力として与える.F に含まれるファイル:f とクエリを包含率を用いたブルームフィルタでファイル:f にクエリ:q の類似コードが含まれる可能性があるか判断する. もしファイル:f がクエリ:q の類似コードを含む可能性があると判断された場合, ファイル:f の各ウィンドウサイズのコードに対して, クエリとの情報距離を計算する. そして情報距離が閾値以下となったコードを類似コードとして判断する. これをすべてのファイル:f に行い, 最終的に報告されたコードをフィルタリングし, それらを類似度順に報告する.

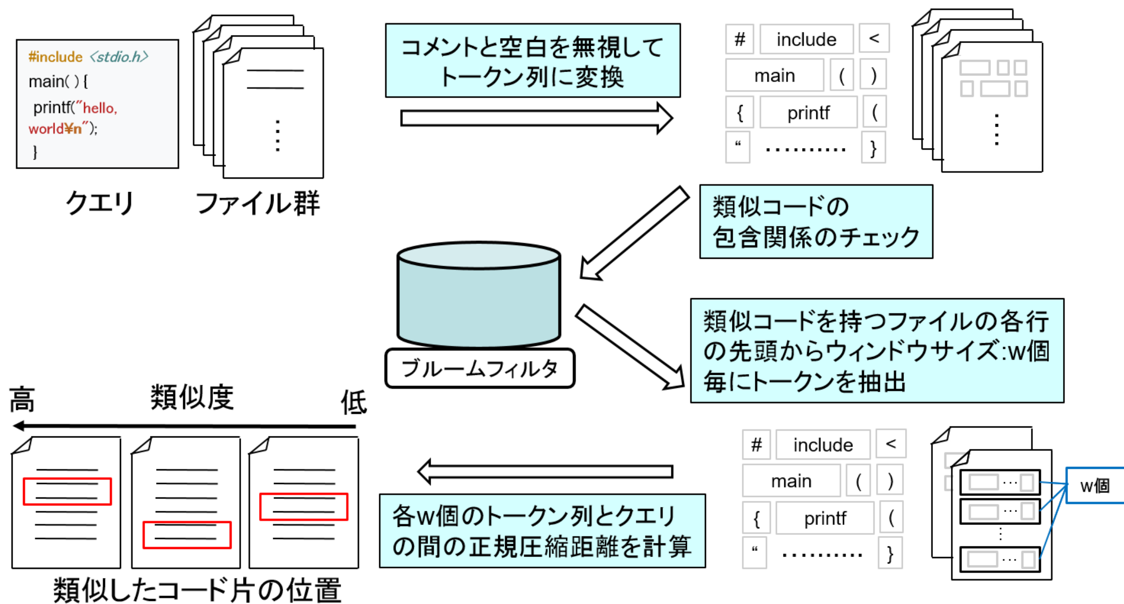

図 7: ブルームフィルタを用いた NCDSearch の処理概要

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適合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. 改変前の改変前の NCDSearch と比較して提案手法によって適合率が下がっていないかを調査する.

RQ3. 実行時間, 適合率を考慮したらめーた最適なパラメータはどんな値か 提案手法は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, ハッシュ関数の数,n-gram の n の値, 包含率の閾値の 4 つのパラメータが存在する.RQ1,RQ2 の調査の過程で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，改変したNCDSearchに対して改変の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

本研究では,3つのリサーチクエスチョンを考察するため，改変したNCDSearchに対して改変のNCDSearchに対して行った実験を同じ条件で行った.2.1.2で記述したが, データセットについて再度説明する. 今回の実験ではPostgreSQL,Git,Linuxという3つのOSSプロジェクトを基にしたCBCDツールのベンチマークデータセットで採用されたデータセットを用いている.[17] このデータセットは各ツールの更新履歴から抽出された完全一致コード, 類似コードを含む53件のクエリ, 検索結果の組の集合から成る. プロジェクトの主なプログラミング言語はC / C ++である. また, クエリには, 次のプロパティがある.

- 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, 最長のクエリには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ち, 残りのクエリはソースコード内に最大13の複数の類似コードを持つ.
- 25個のクエリが合, ソースコード内に完全一致のコードを持つ.

表4.1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, ファイルのサイズを示している.

Projects	#Queries	#Bugs	Median #Files	Median 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

このデータセットに対して各ツールを使用することで以下の指標を扱う. 適合率 (recall), 実行時間を調査した.

4.2 NCDSearchの設定

今回の評価実験は改変前との比較が目的のため, 改変前の評価実験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recall

を示した Deflate を使用している. また, 表 4.2 が本実験に用いるパラメータである. またブルームフィルタで使用するビット列のビット数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 [19].

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

4.3 実験の考察

4.3.1 RQ1. 改変前に比べて NCDSreach の実行時間は削減されるのか

まず実験的にビット数は十分に大きい数, 検索対象の集合の要素数の 1000 倍に設定しを, ハッシュ関数を 1 つ, そしてパラメータとして n-gram の値を 1-gram から 20-gram, 包含率の閾値を 0.0-1.0 の範囲で実験を行った. 図 8 がデータセットに対する実行時間の結果である.

この結果を見ると, 全てのデータセットで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ている. また, 図**??**, 図**??**, 図**??**が各データセットに対する実験結果であり, 全てのデータセットで全体の結果と同じように最大で 9 割もの実行時間を削減できている. 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.5-gram 以上の n-gram で Θ が 0.5 以上のものはすべて 9 割りもの実行時間を削減できている. 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, ほとんどのファイルに情報距離の計算を行い, ブルームフィルタの計算も加わったためである. これが理由で 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.

4.3.2 RQ2. ブルームフィルタの実装によって適合率を下げることはないか

RQ1 の実験と同じ条件で, ビット数は十分に大きい数, 検索対象の集合の要素数の 1000 倍に設定しを, ハッシュ関数を 1 つ, そしてパラメータとして n-gram の値, 包含率の閾値をパラメータとして実験を行った. 図 12 がその実験の実行時間の結果である. この結果を見るとこの結果を見ると, パラメータにはよるが改変前と同じように recall が 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 recall が下がっている. これは n-gram

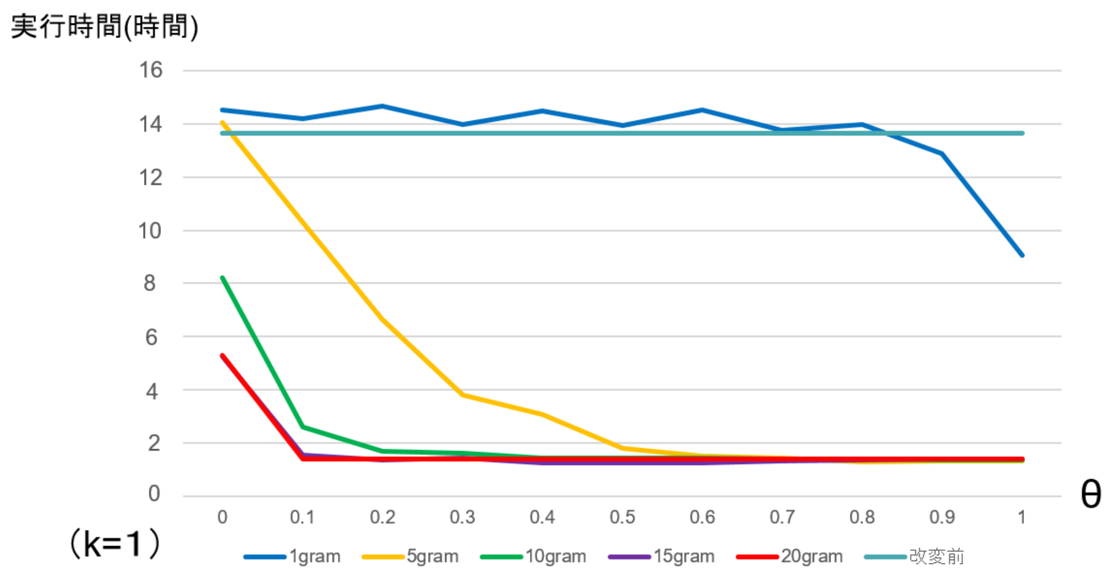

図 8: 全体の実行時間

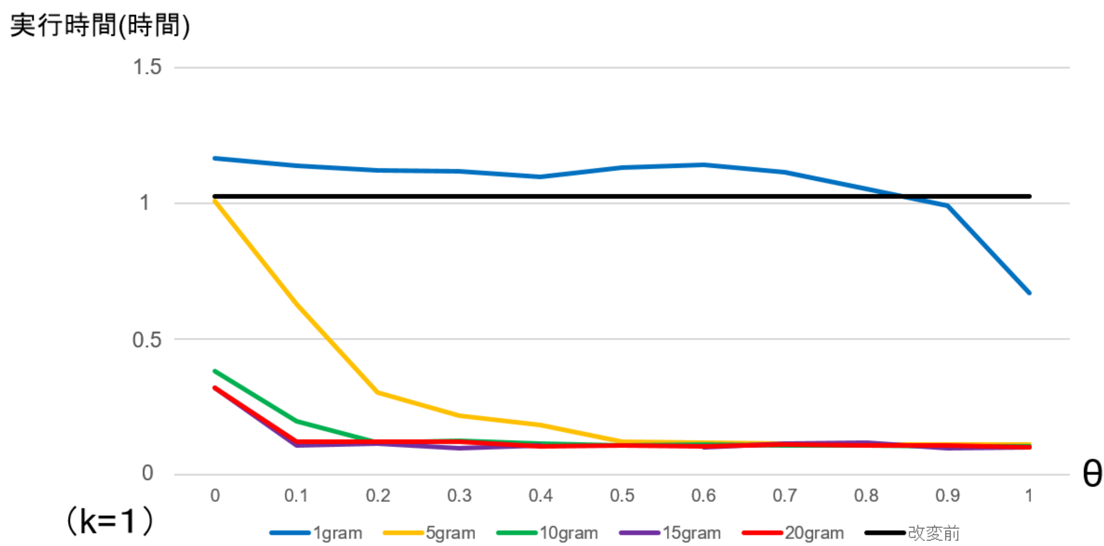

図 9: PostgreSQL に対しての実行時間

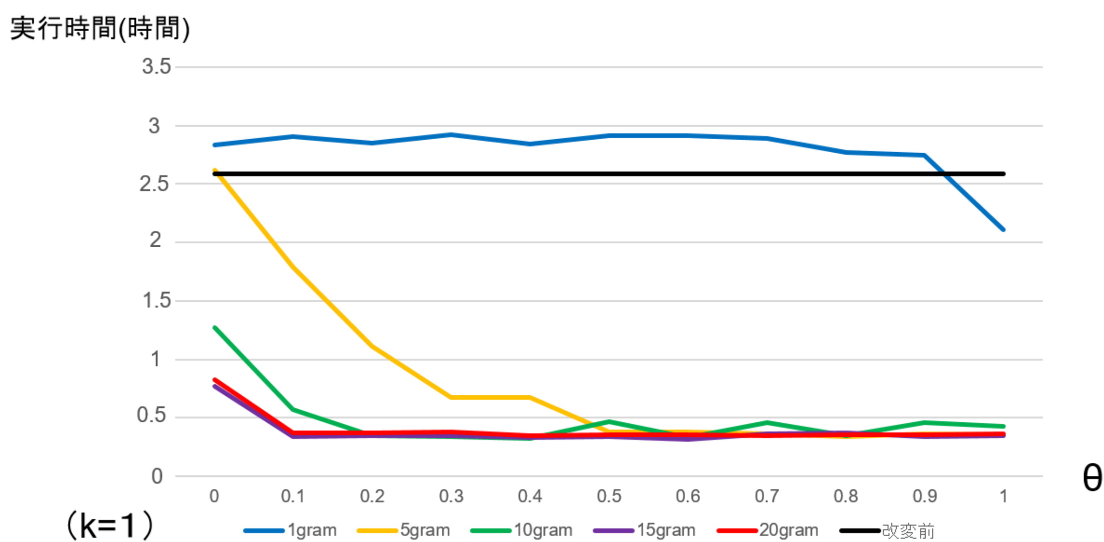

図 10: Git に対しての実行時間

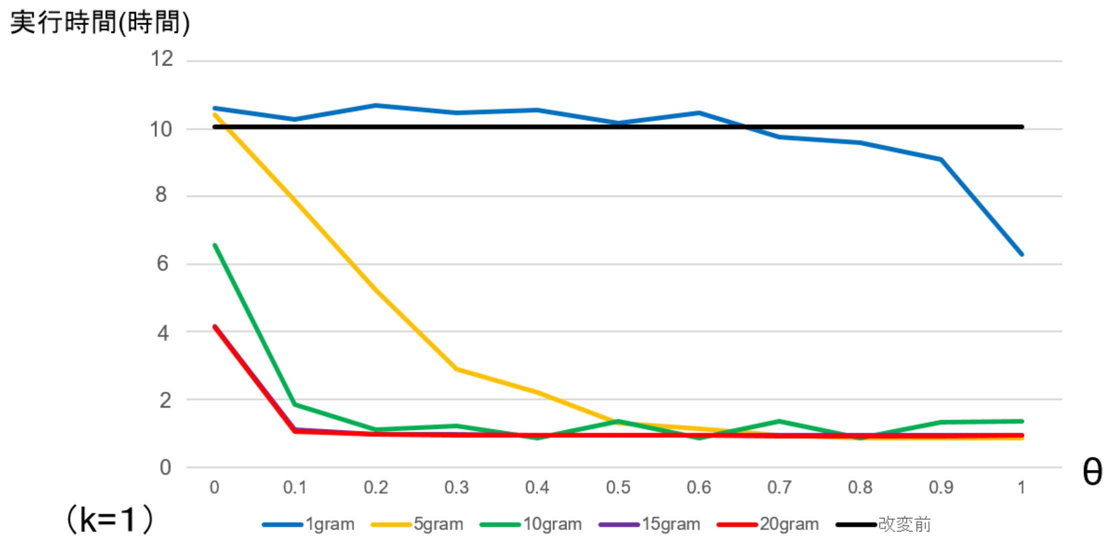

図 11: Linux に対しての実験結果

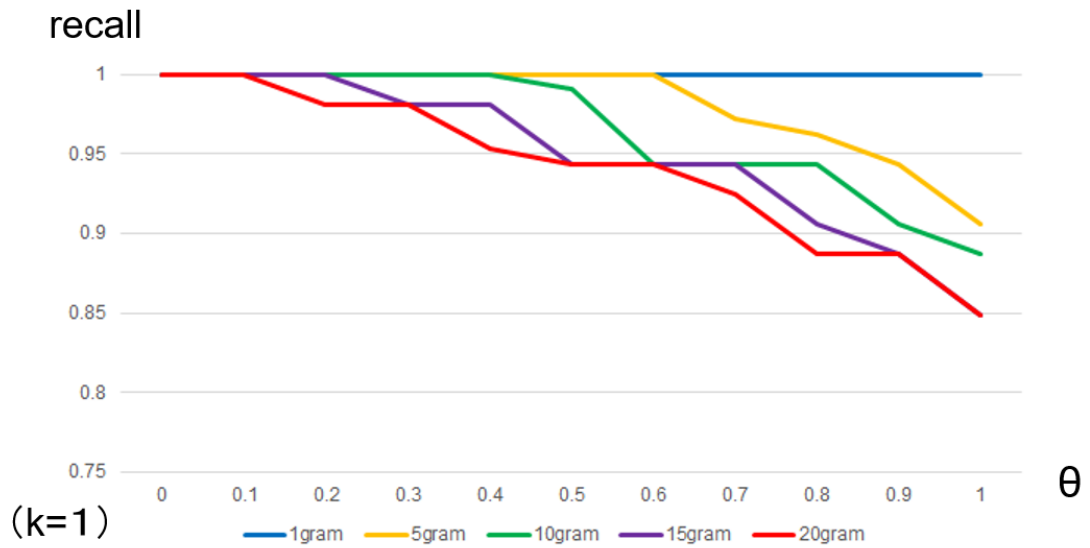

図 12: RQ2 に対しての実験結果

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, 包含率も下がるためである. しかし 20-gram までは recall が 1 となるパラメータが存在する.

4.3.3 RQ3. 最適なパラメータはどんな値か

RQ1,RQ2 の実験結果を見ると,9 割もの実行時間を削減でき, 同時に recall が 1 となっているパラメータが存在し, そのパラメータを表 5 に示す. この表を見ると,10-gram が最もパ

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

表 5: 理想のパラメータ

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-gram は 10-gram に設定し,0.2 − 0.4 の中間の値である $\Theta=0.3$ の時が最も安定的に recall 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.

4.3.4 ハッシュ関数

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, 包含率の閾値 Θ をパラメータとして実験を行った. その結果が図 13, 表 6 である.

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だと考えられる. また,recall 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,recall が高くなるので, ハッシュ関数の数は 1 つが理想的である.

表 6: RQ3：ハッシュ関数に対しての実験結果（recall）

	0	0.2	0.4	0.6	0.8	1.0
K=1	1	1	1	0.943396	0.943396	0.886792
K=5	1	1	1	0.943396	0.924528	0.886792
K=10	1	1	1	0.943396	0.924528	0.886792

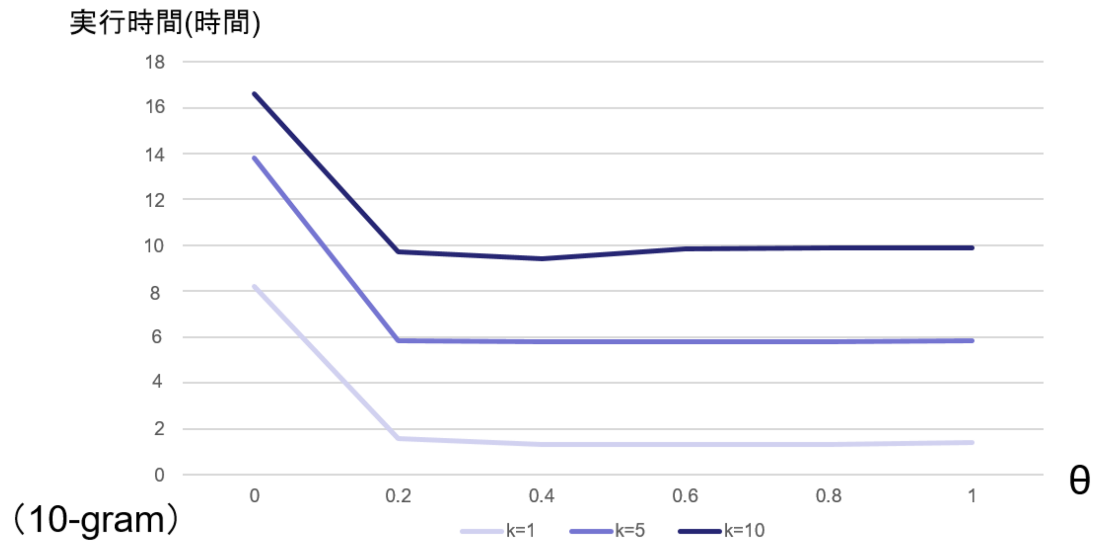

図 13: RQ3:ハッシュ関数に対しての実験結果（実行時間）

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ベースを使用し, 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. その上で本手法の有用性, 最適なパラメータの検証を行っている. そのため, 他のデータベースで同じようにパラメータで実験しても本研究の結果と同様の結果が得られない可能性がある. また評価実験の試行回数は 1 回のみであるため, 再度実験を行った場合, 本研究の実験結果と異なる可能性がある.

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. 提案手法ではブルームフィルタに包含率を用いることで, 完全一致のコードだけでなく類似コードの検出も可能にした. 包含率には n-gram を使用することで, 高い効率で類似コードを検出することを可能にした. また, 既存の NCDSearch に対して行われた評価実験を提案手法にも行い, 改変前との比較評価を行った. その結果, 既存手法の NCDSearch と比較して, 適合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.

謝辞

本研究，ならびに研究室生活，研究への心構えなど，多岐にわたって常に御指導および御助言を賜りまし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克郎教授に心より深く感謝いたします．

本研究において，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適切な御指導および御助言を賜りました松下准教授に深く感謝いたします．

本研究において，研究の方針や論文の書き方など，様々な御指導を頂きました奈良先端科学技術大学院大学先端科学研究科石尾隆准教授に深く感謝いたします．

本研究にならびに研究室生活において，技術的な面から日常的な面まで様々な御指導および御助言を頂きまし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 神田哲也 助教に深く感謝いたします．

本研究に限らず，様々なご指導およびご助言をいただきまし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 嶋利一真氏に感謝いたします．

最後に，その他様々な御指導，御助言等を頂い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] 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] 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.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[8] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 172–181. IEEE, 2008.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Nils Göde and Rainer Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pp. 219–228. IEEE, 2009.

[16] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul Vitányi. The similarity metric. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pp. 863–872, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[17] 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.

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

[19] 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.