

SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration

Guanhua Li, Yijian Wu, Chanchal K. Roy, Jun Sun, Xin
Peng, Nanjie Zhan, Bin Hu and Jingyi Ma

*2020 IEEE 27th International Conference on Software Analysis, Evolution
and Reengineering (SANER), 2020, pp. 272-283*

背景

- ビッグデータの時代には, クローン検出ツールは大規模なプロジェクトリポジトリに対応必要がある
- ただし, ビッグコード内のクローンを効率的に検出することは依然として困難
- 関数の粒度でも, 通常, 2億5000万行のコードが持っているプロジェクトにはクローンを検出するのが数日かかる
- より細かい粒度でクローンを検出する (小さいセグメントクローンなど) のははるかにコストがかかる

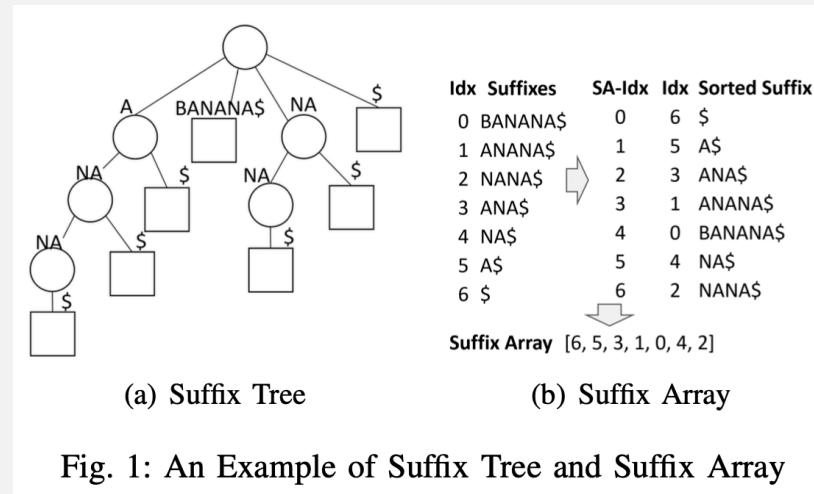
背景

- クローン検出のパフォーマンスに不可欠ものは：
 - 適切なコード表現
 - マッチング構造
- トークンベースの変換と接尾辞木または接尾辞配列のマッチングの組み合わせは合理的
- ただし, 大規模なコードリポジトリで効率的に実行できるかやタイプ3クローンがきちんと検出できるかという疑問がある

目的

- SAGA : Suffix Array based clone detection tool with in-built GPU Acceleration
- セグメントの粒度でType-1/2/3クローンを検出する
SAGAを提案する
- 効率, スケーラビリティ, 再現率, 適合率, 一貫性の観点からSAGAを評価する

接尾辞木とは



- 接尾辞木：文字列のすべての接尾を表す辞書木
- 接尾辞配列：
 - 文字列のインデックス値を使用して接尾辞を表す
 - 接尾辞木の代替表現

問題点 (1)

接尾辞木の構築には大量のメモリが必要,
接尾辞配列の構築には大量の計算が必要

- Data Parallel Prefix-Doubling (DPPD) アルゴリズムを利用し、接尾辞配列を構築する

問題点 (2)

セグメントの粒度でタイプ3クローンを検出することは難しい

- 隣接するタイプ1/2クローンをタイプ3クローンにマージする

問題点 (3)

接尾辞木や接尾辞配列は大きすぎることになる恐れがある

- GPU計算の設定にデータチャンクの観念を適用する
 - データチャンク：データを何個かに分ける
- 次の2つの最適化を適用する
 1. マルチGPUサポート
 - 各データチャンクは個別に処理できるため, 複数のGPUで同時に実行できる
 2. CPU-GPUアーキテクチャのパイプライン処理
 - CPU：前処理と変換を操作する
 - GPU：接尾辞配列の構築とマッチングの検出を操作する
 - CPU-GPUアーキテクチャを最大限に活用し, パフォーマンスをさらに向上させる

SAGAによるタイプ3のコードクローン検出方法

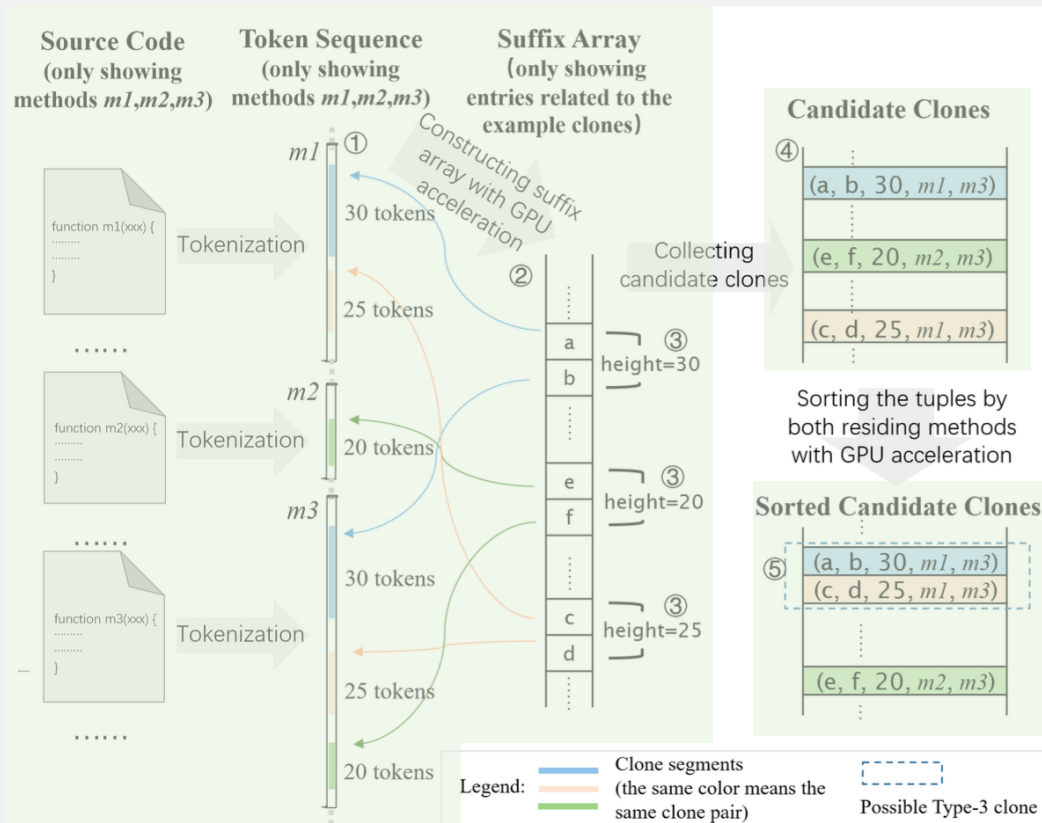


Fig. 3: Detecting Type-3 Clones: An Example

- 1) ソースコードは最初に一連のトークンに変換される (①)
- 2) 接尾辞配列に変換する (②)
- 3) 各候補クローンを5タプル (seg1, seg2, len, method1, method2) として示す (④)
- 4) 2つのコードセグメントの両方に存在するメソッドに従って候補クローンを並べ替える (⑤)
- 5) 候補クローンをソートした後、隣接する候補クローンをマージしてタイプ3クローンを形成する

※詳細は補足資料で

アプローチ

- SAGAの実行時間, 再現率, 適合率を評価
 - BigCloneBench, Mutation and Injection Framework, および10億行のコードで構成されるオープンソースコードベースを使用して, 他のクローン検出ツールと比較し, 分析する
- 予備的なケーススタディを実施する
 - SAGAを使用して85,202のプロジェクト
 - 大きなコード分析におけるその有用性を確認

結果

- A. 実行時間とスケーラビリティ
- B. 再現率
- C. 適合率
- D. ツール間の一貫性
- E. 予備的なケーススタディ

実行時間とスケーラビリティ

- データセット：
 - IJaDataset - 約2億5000万行のJavaソースコードが含まれている
 - GitHubからダウンロードされたJavaプロジェクト
 - 約8,000のプロジェクト(3億 LOC) がある
- GitHubからダウンロードされたプロジェクトとIJaDatasetを組み合わせると, 5億5000万行のデータセット
- このデータセットはスケーラビリティ実験と再現率の実験で使用された

実行時間とスケーラビリティ

TABLE II: Tool Settings for Scalability Test

Tool	Settings
SAGA	Min clone length 50 tokens, min candidate length 20 tokens, min similarity 0.6, function granularity
SourcererCC	Min length 6 lines, min similarity 0.7, function granularity
CloneWorks	Min length 6 lines, min clone length 50 tokens, min similarity 0.7, function granularity
CCFinder	Min clone length 50 tokens
NICAD	Min clone length 6 lines, identifier abstraction none, similarity 0.7
Deckard	Min clone length 50 tokens, similarity 0.85, 2 token stride

- 環境と設定：
 - ソースコードデータセットに対して, SourcererCC, CCFinderX, NICAD, Deckard, CloneWorksおよびSAGAを実行する
 - すべての実験: Intel i7 CPU (8コア、16スレッド), 32GB RAM, 1TB SSD, および6GBグラフィックメモリを備えた1枚のNVIDIA GTX1080Tiグラフィックカードで実行される
 - 実行時間はiClone, CCAAligner, Oreoと比較する
 - これらのツールは手元にない
 - iClone - 文献で報告されている実行時間を要した
 - CCAAlignerとOreo - SourcererCCの実行時間と比較して, 文献に基づいてこれら2つのツールの実行時間を見積もる

実行時間とスケーラビリティ

TABLE III: Execution Time

LOC	SAGA	SourcererCC	CCFinder	NICAD	Deckard	iClones	CCAligner	Oreo	CloneWorks(C)	CloneWorks(A)
1K	1s	3s	2s	1s	1s	1s*	1s**	–	3s	3s
10K	1s	5s	5s	1s	4s	1s*	1s**	–	3s	3s
100K	4s	7s	10s	5s	32s	2s*	3s**	–	4s	4s
1M	9s	37s	39s	12s	27m12s	MEM*	>20s**	–	26s	26s
10M	59s	12m21s	6m30s	19m49s	ERROR	–	>5m**	–	11m12s	11m3s
100M	10m57s	12h27m	9h49m	ERROR	–	–	–	–	2h19m	1h55m
250M	1h24m	36h**	>25h**	–	–	–	–	>222h**	18h27m	8h3m
550M	4h50m	–	–	–	–	–	–	–	–	–

* Data reported in the literature

** Data estimated based on reported time with consideration of environment differences

- 実行時間の結果を表3に示す
- SAGAは, 1億 LOCソースコードのクローン検出を10分57秒で完了する
 - SourcererCCおよびCCFinderの実行時間より約50～70倍短い
 - CloneWorksの約12倍の速度
- 1億LOCのスケールでは, NICAD, DeckardおよびiClonesはコードの解析に失敗するか, メモリ不足エラー報告された

再現率

TABLE IV: Recall and Precision on BigCloneBench

Tool	Threshold/Setting	T1(35,800)		T2 (4,574)		VST3 (4,156)		ST3 (15,013)		MT3(79,922)		WT3/T4(7.75M)		Precision	
		%	#	%	#	%	#	%	#	%	#	%	#	Func	Sgmt
SAGA	sim=0.6, $l_{mc}=40$, function	100	35,781	100	4,552	95	3,932	60	9,030	10	7,686	0	5,936	99	–
SAGA	sim=0.7, $l_{mc}=40$, function	100	35,781	98	4,504	91	3,769	45	6,715	5	4,141	0	2,171	99	–
SAGA	sim=0.6, $l_{mc}=50$, function	99	35,552	96	4,375	93	3,866	56	8,396	8	6,055	0	3,126	99	–
SAGA	sim=0.7, $l_{mc}=50$, segment	–	–	–	–	–	–	–	–	–	–	–	–	–	75
SourcererCC*	sim=0.7, MIT=1	100	35,797	98	4,462	93	3,871	61	9,099	5	4,187	0	2,005	98	–
SourcererCC+	sim=0.7, MIL=6	98	35,185	85	3,849	89	3,680	58	8,651	7	5,211	1	80,520	83	–
CCFinder*	MIT=50	100	–	93	–	62	–	15	–	1	–	0	–	–	72
CloneWorks(A)*	sim=0.7, MIT=1	100	35,777	99	4,544	98	4,090	93	13,976	3	2,700	0	35	99	–
NICAD*	sim=0.7, MIL=6	100	35,769	99	4,541	98	4,091	93	13,910	1	671	0	12	99	–
CCAligner*	sim=0.7, $q = 6$, $e = 1$	100	–	99	–	97	–	70	–	10	–	–	–	80	–
Oreo*	sim=0.55, MIT=15, $\tau=0.6$	100	35,798	99	4,547	100	4,139	89	13,391	30	23,834	1	57,273	90	–

* Data reported in the literature (same configuration)

+ Data collected in the authors' replication experiment

- BigCloneBenchとMutation and Injection Frameworkの2つのベンチマークを使用する
- SAGAは, Type-1 / 2クローンだけでなく, Type-3クローンも高い再現率が持っている

適合率

TABLE IV: Recall and Precision on BigCloneBench

Tool	Threshold/Setting	T1(35,800)		T2 (4,574)		VST3 (4,156)		ST3 (15,013)		MT3(79,922)		WT3/T4(7.75M)		Precision	
		%	#	%	#	%	#	%	#	%	#	%	#	Func	Sgmt
SAGA	sim=0.6, $l_{mc}=40$, function	100	35,781	100	4,552	95	3,932	60	9,030	10	7,686	0	5,936	99	–
SAGA	sim=0.7, $l_{mc}=40$, function	100	35,781	98	4,504	91	3,769	45	6,715	5	4,141	0	2,171	99	–
SAGA	sim=0.6, $l_{mc}=50$, function	99	35,552	96	4,375	93	3,866	56	8,396	8	6,055	0	3,126	99	–
SAGA	sim=0.7, $l_{mc}=50$, segment	–	–	–	–	–	–	–	–	–	–	–	–	–	75
SourcererCC*	sim=0.7, MIT=1	100	35,797	98	4,462	93	3,871	61	9,099	5	4,187	0	2,005	98	–
SourcererCC+	sim=0.7, MIL=6	98	35,185	85	3,849	89	3,680	58	8,651	7	5,211	1	80,520	83	–
CCFinder*	MIT=50	100	–	93	–	62	–	15	–	1	–	0	–	–	72
CloneWorks(A)*	sim=0.7, MIT=1	100	35,777	99	4,544	98	4,090	93	13,976	3	2,700	0	35	99	–
NICAD*	sim=0.7, MIL=6	100	35,769	99	4,541	98	4,091	93	13,910	1	671	0	12	99	–
CCAligner*	sim=0.7, $q = 6$, $e = 1$	100	–	99	–	97	–	70	–	10	–	–	–	80	–
Oreo*	sim=0.55, MIT=15, $\tau=0.6$	100	35,798	99	4,547	100	4,139	89	13,391	30	23,834	1	57,273	90	–

* Data reported in the literature (same configuration)

+ Data collected in the authors' replication experiment

- 適合率は, ランダムで統計的に有意なサンプルを検証することによって手動で測定される
- 結果は, SAGAが99%の適合率を持っていることを示している

ツール間の一貫性

- 結果が一貫しているかどうかを確認するために, **SAGA**, **SourcererCC**, および **CCFinder**によって報告された実際のクローンを比較する (5つのプロジェクトを利用)
- 比較した結果：
 - **SourcererCC** : 6,749のクローンペア
 - **SAGA** : 9,548クローンペア
 - **CCFinder** : コードの一部のみを個別に検出するため、検出されるクローンの数ははるかに少ない
- 一般に, **SAGA** :
 - 関数の粒度で**SourcererCC**と大体同じ結果が出た
 - セグメントの粒度でクローン検出できる

予備的なケーススタディ

- 85,202のGitHub Javaプロジェクトを利用して、予備的なケーススタディを実施した
- 以前の研究は、小さなコードベース(6,000未満のJavaプロジェクト)だけ対応できるという制限があった
- SAGAの設定：
 - クローンの最小の長さ(lmc) = 50
 - 候補クローンの最小の長さ(lmcc) = 20
 - 類似性 = 0.7
- 結果：約3億2700万のクローンコードセグメントを含む1億4200万を超えるクローングループを取得
- SAGAは、大規模なクローンの検出と分析、および大規模なコードリポジトリに対応できる有望なツールであることを示している

まとめ

- ビッグコードのセグメント粒度でタイプ3クローンを検出することの難しさに取り組み, **GPU**計算能力を活用して検出プロセスを高速化する接尾辞配列ベースのコードクローン検出ツールである **SAGA**を提案する
- このツールは, **11**時間の実行時間内に最大**10**億行のコードが処理できる
- **SAGA**は大規模なデータが処理でき, セグメントレベルの粒度でニアミスクローンを検出できる唯一のツール

所感

- 大きなデータが処理できるように**GPU**を最適化するのは良い観点と思った
- **SAGA**の実際に実行ステップを説明してほしい

補足資料

SAGAによってタイプ3のコードクローンを検出する方法

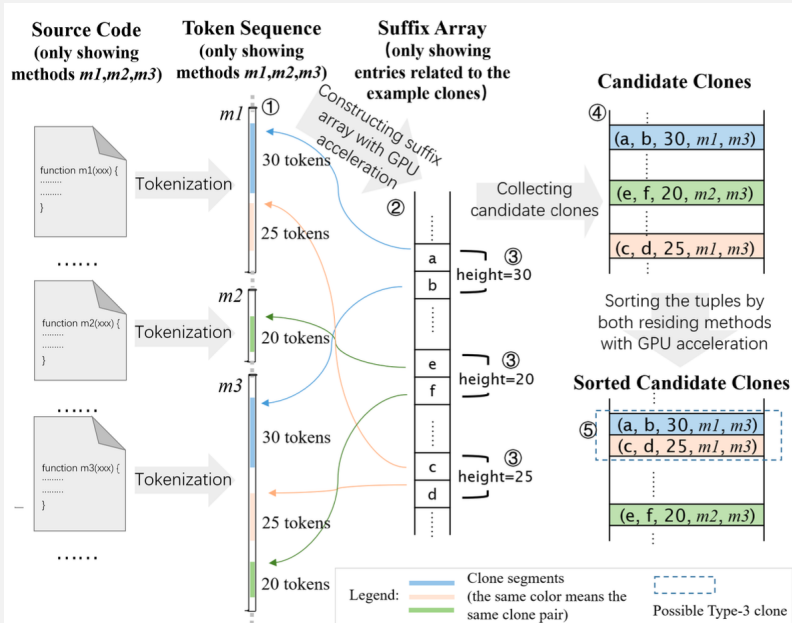


Fig. 3: Detecting Type-3 Clones: An Example

- 1) ソースコードは最初に一連のトークンに変換される(図3に①として示されている)
- 2) 接尾辞配列に変換する(②で表示)
- 3) 各候補クローンを5タプル(seg1, seg2, len, method1, method2)として示す(④として表示)
 - seg1とseg2はコードセグメント
 - lenはクローンの長さ
 - method1とmethod2は、コードセグメントが存在するメソッド
 - $len < lmc$ - 候補クローンがタイプ3クローンの検出に使用される
 - $lmc < len$ - 候補クローンはタイプ1/2クローンとして直接報告される
- 4) 2つのコードセグメントの両方に存在するメソッドに従って候補クローンを並べ替える(⑤)
 - 安定ソートAPI (stable_sort_by_key) を利用
- 5) 候補クローンをソートした後、隣接する候補クローンをマージしてタイプ3クローンを形成する

Algorithm 1: Detecting Type-3 Clones

Input: The sorted array $ccArray$ of the 5-tuples of all candidate clones

Output: *Result*: Set of Type-3 clones

```
1  $Result = \emptyset$ ;  
2  $cc =$  the first candidate clone in  $ccArray$ ;  
3  $ccToMerge = cc$ ; // a running  $cc$  for merge  
4 while  $cc \neq null$  do  
5    $ccNext =$  next candidate clone in  $ccArray$ ;  
6   if  $IsMergeable(ccToMerge, ccNext)$  then  
7      $cc = Merge(cc, ccNext)$ ;  
8      $ccToMerge = ccNext$ ;  
9   end  
10  else  
11    if  $cc.length \geq l_{mc}$  then  
12      Add  $cc$  to  $Result$ ;  
13    end  
14     $cc =$  next candidate clone in  $ccArray$ ;  
15     $ccToMerge = cc$ ;  
16  end  
17 end  
18 return  $Result$ ;
```

タイプ3のコードクローンを検出するために利用されているアルゴリズム

Algorithm 2: Detect if candidate clones are mergeable

Input: Two candidate clones cc_1, cc_2

Output: true iff cc_1 and cc_2 are mergeable

```
1 Function  $IsMergeable(cc_1, cc_2)$   
2   if  $cc_1.method_1 \neq cc_2.method_1$  or  
3      $cc_1.method_2 \neq cc_2.method_2$  then  
4       return false  
5   end  
6    $gap = \max(gap \text{ between } cc_1.seg_1 \text{ and } cc_2.seg_1, gap$   
7      $\text{ between } cc_1.seg_2 \text{ and } cc_2.seg_2);$   
8   if  $gap > (cc_1.length + cc_2.length + gap) * (1 - \theta)$   
9     then  
10    return false  
11  end  
12  return true
```

候補クローンがマージできるかどうかを検出するアルゴリズム