POPLミーティング
(2005/10/26)

米澤研究室 M1 佐藤秀明

概要

- ソースコードの類似性を用いたさまざまな解析
  - 剽窃を検知
  - コードの複製を発見 (メインの話)
  - ソフトウェアの進化系統図を構築
  - ソースコードの作者を特定

概要

- ソースコードの類似性を用いたさまざまな解析
  - 剽窃を検知
  - コードの複製を発見 (メインの話)
  - ソフトウェアの進化系統図を構築
  - ソースコードの作者を特定

ソースコードの剽窃

- 他人のソースを自分のものとして利用
  - 大学のプログラミング演習で
  - 商用ソフトウェア開発で
- ソースの類似度を計測してチェック
  - 「似ている」の定義とは？

Attribute Counting Systems

- プログラム中の各要素の個数を比較 [1]
  - ユニークな演算子 / オペランドの数
  - 全体 / コメントの行数
  - …
- 各々の数値が近ければ「似ている」とみなす
  - これでは明らかに不十分

Structure Metric Systems

- トークン列のアラインメント問題に落とす [1]
  - 最長共通部分文字列
  - Dynamic programming によるスコアリング
- コードの転位 (位置の入れ替わり) に弱い

Shared Information(1/2)

- 改造 LZ 法による圧縮サイズを比較 [2]
  - LZ 法： 既に出現した文字列 a と同じ b は、
    - a へのポインタ
    - a の長さ

    の組で置換
  - x と y が等しいなら、x と xy の圧縮後サイズは等しい

Shared Information(2/2)

- 学生課題を分析
  - 数百人規模のテストを数回
  - Structure metric systems 以上の検知力
    - 新たな剽窃を発見
    - ついでに学内カップルも発見
      - 剽窃自体は頑として認めず
- Random insertion に対する耐性

概要

- ソースコードの類似性を用いたさまざまな解析
  - 剽窃を検知
  - コードの複製を発見 (メインの話)
  - ソフトウェアの進化系統図を構築
  - ソースコードの作者を特定

Code Clone

- 機能または文面が似ているコード
  - ソースをコピー & ペースト
  - 設計の洗練不足
- 大規模なシステム開発において有害
  - メンテナンス性の低下
- ソースコードの類似度を用いて発見

剽窃検知との違い

- 元ソースの情報をつぶしてはいけない
  - 似ている箇所を指摘する必要性
  - 類似度を示す数値だけでは意味がない
- いろいろな方法が存在
  - Token-based analysis
  - AST
  - Program dependence graph
  - Higher-level similarity patterns

言語非依存なアプローチ [3](1/5)

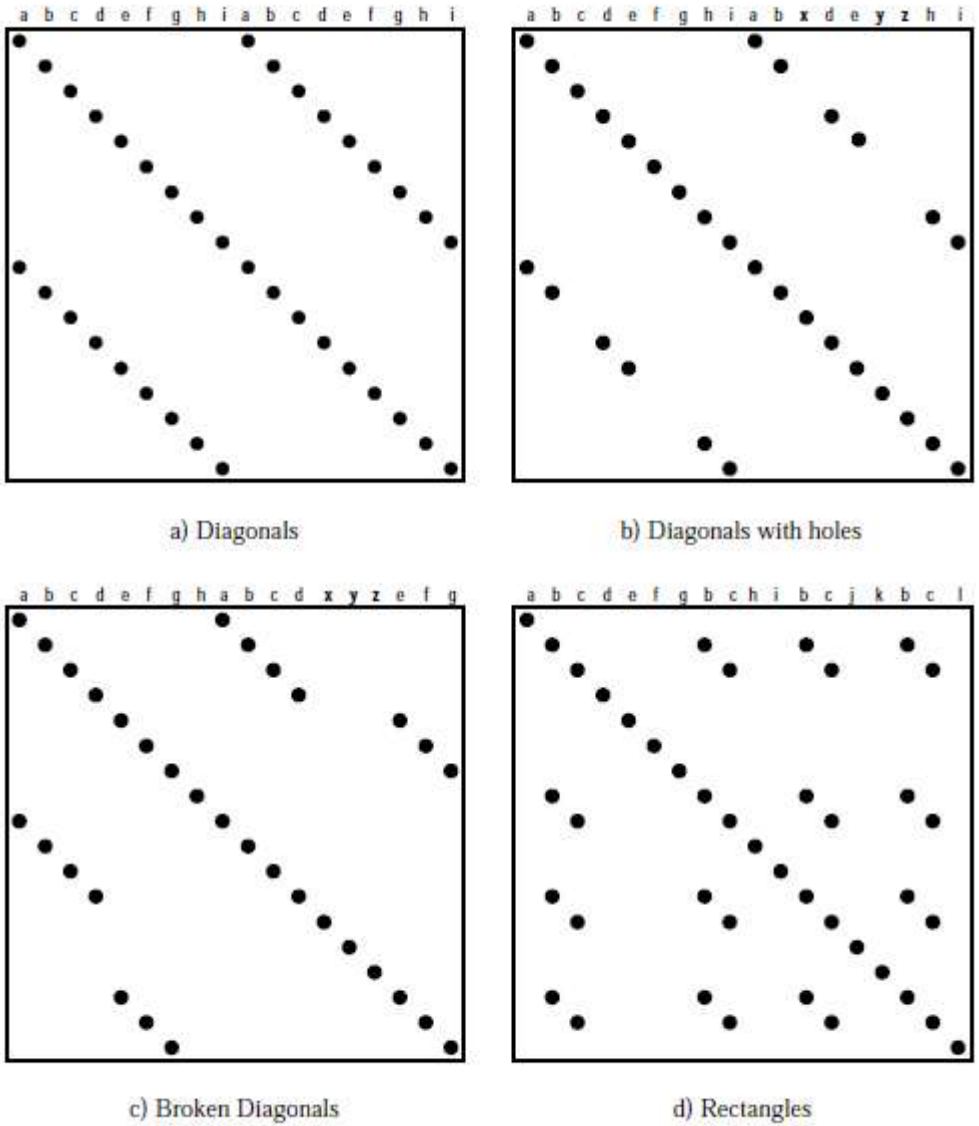
- 構文解析しない
  - 処理が軽い
  - さまざまな言語に適用可能
- 各行ごとの完全マッチで評価
  - 処理が軽い
- 比較行列を作成
  - マッチした部分を視覚的に表現

言語非依存なアプローチ (2/5)

- 比較行列
  - row と column はファイルの行ナンバー
  - マッチングに対応する成分の値は 'true'
- 比較行列の性質
  - 必ず対称行列になる
  - 対角成分は常に 'true'

言語非依存なアプローチ (3/5)

- 比較行列の例
  - a コードクローン
  - b 一部変更あり (hole)
    - 小さな hole なら無視してクローンとみなす
  - c 一部追加 / 削除あり
  - d 周期的な出現
    - switch 中の break など



言語非依存なアプローチ (4/5)

- 評価に用いたプログラム

Case	*Language*	*Size*	*# Files*	*LOC*
gcc	C	13.4 Mb	221	460000
Database Server	Smalltalk	7.1 Mb	593	245000
Payroll	Cobol	3 Mb	13	40000
Message Board	Python	265 K	36	6500

  - gcc: コードクローンが少ないらしい
  - Database server と Payroll: クローンが沢山あるらしい
    - 共に商用
  - Message Board: 事前情報なし
    - public domain

言語非依存なアプローチ (5/5)

- 発見したクローンの数…予想通り

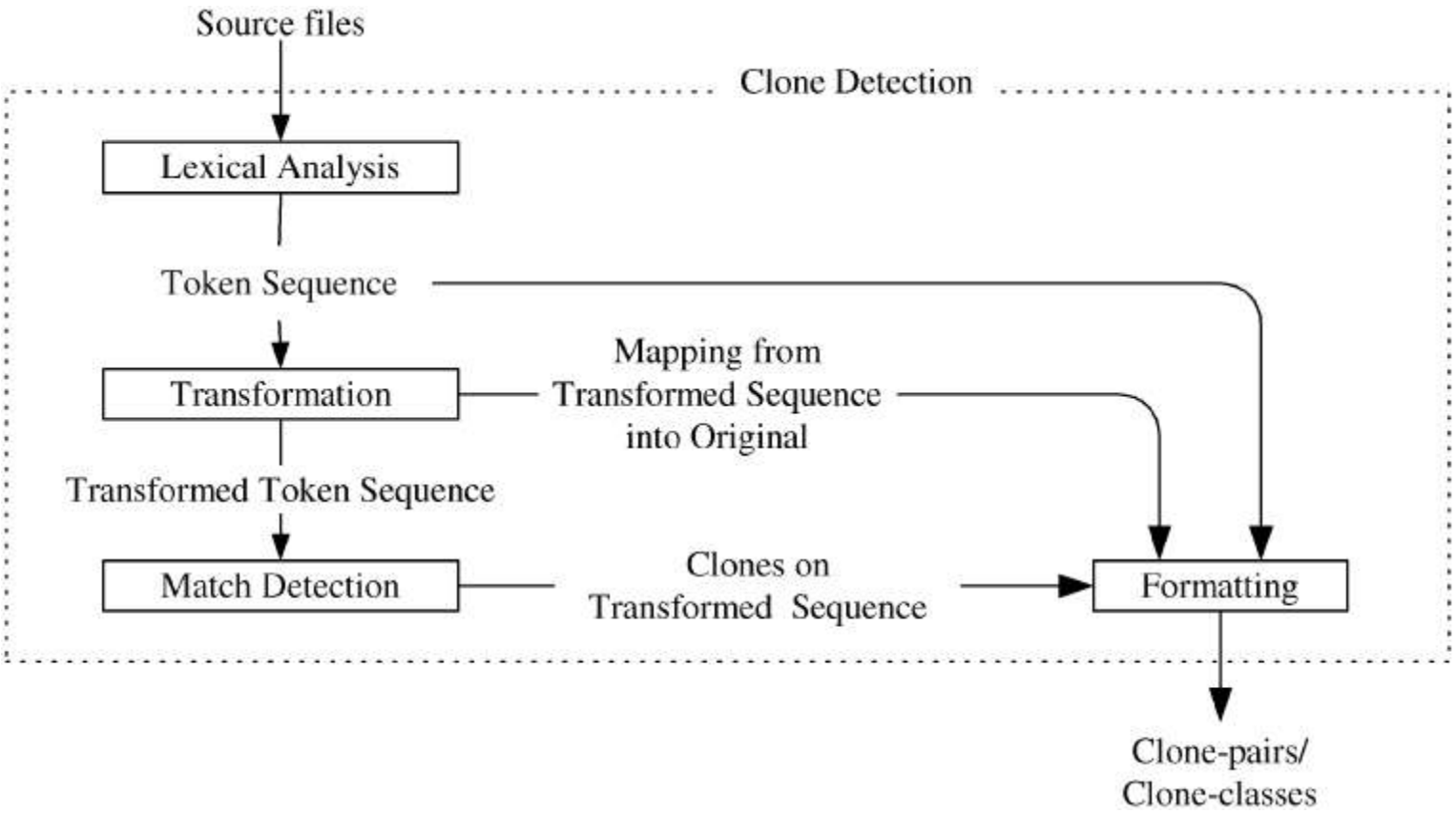
Average percentage of duplication found per file				
Case	gcc	Datab. S.	Payroll	Mess. B.
effective LOC	8.7%	36.4%	59.3%	29.4%
entire LOC	5.9%	23.3%	25.4%	17.4%
# of files with duplication	143	464	13	24
Total # of Files	170	593	13	36

- 実行時間…結構かかる

Performance Statistics				
Case	gcc	Database S.	Payroll	Message B.
# of files	221	593	13	36
LOC	460000	245000	40000	6500
parallel tasks	1	2	1	1
running time	6h30m	5h/7h	8m	1m
comparisons	1670	22939	51	50

CCFinder[4](1/5)

- トークン列のマッチング
- 各言語固有の処理を最小化
- 変数名のバリエーションを吸収



CCFinder(2/5)

- 言語依存の Transformation rule
  - identifier の一般化
  - プログラム構造の明確化
- 次頁は Java の例

CCFinder(3/5)

Rule #	Rule description
RJ1 Remove package names	**(PackageName '.')+ ClassName → ClassName** Here, PackageName is a word that begins with a small letter and ClassName is a capitalized word. In Java source files, a class is referred to with either the full package name or a shorter name by using import sentences. The transformation is to neglect the attribution so that they are considered equivalent in clone detection. Ex. `java.lang.Math.PI` is transformed to `Math.PI`.
RJ2 Supplement callees	**NDotOrNew NClassName '(' → NDotOrNew CalleeIdentifier '.' NClassName '('** Here, NDotOrNew is a token except '.' or 'new'. NClassName is an uncapitalized word. CalleeIdentifier is a token for an omitted callee. By language specification a method is either an instance method or a class method. Therefore, if an instance calls a method without a callee instance or class then the omitted callee is the instance itself or a class of it.
RJ3 Remove initializtion lists	**'=' '{' InitalizationList, '}' → '=' '{' '}'** **']' '{' InitalizationList, '}' → ']' '{' '}'** Here, InitalizationList is a sequence of Name, Number, String, Operators, ',', '(', ')', '{', and '}'. These rules are an expansion of rule RC3. The second rule is applied where an array is created with initialization by a new expression. For example, `return new int[] { 1, 2, 3 };`.
RJ4 Separate class definitions	**Insert UniqueIdentifier at each end of the top-level definitions and declaration.** This rule prevents extracting clone pairs of the code portions that begin at the middle of a class definition and end at the middle of another class definition.
RJ5 Remove accessibility keywords	**AccessibilityKeyword → *Φ*** Here, phi is a null sequence. Ex. `protected void foo()` is translated to void `foo()`.
RJ6 Convert to compound block	**Each single statement after if (), do, else, for (), and while () is transformed to a compound block.** Ex. `if (a == 1) b = 2;` is transformed to `if (a == 1) { b = 2; }`

CCFinder(4/5)

- 変数名の置換を行うTransformation rule
  - 同一の special case に置換

```
 1| $p $p ($p $p & $p ) {
 2| $p $p = $p ;
 3| $p $p
 4| = $p . $p ( ) ;
 5| for ( ; $p != $p . $p ( ) ; ++ $p ) {
 6| $p << $p << $p
 7| << * $p << $p ;
 8| ++ $p ;
 9| }
10| }
11| $p $p ($p $p & $p ) {
12| $p $p = $p ;
13| $p $p
14| = $p . $p ( ) ;
15| for ( ; $p != $p . $p ( ) ; ++ $p ) {
16| $p << $p << $p
17| << $p -> $p << $p
18| << $p -> $p << $p ;
19| ++ $p ;
20| }
21| }
```

CCFinder(5/5)

- 評価

Effects of Transformation Rules and Other Preprocessing Techniques for JDK

	P+R+123456	*P+12456*	*P+R+34*	*R*
# clone pairs	8047	51825	8063	3842
# clone classes	2333	4638	2324	1038
COVERAGE (% LOC)	21.35%	29.15%	20.51%	8.76%
COVERAGE (% FILE)	46.24%	53.38%	45.34%	27.49%

"P+R+123456' means that parameter replacement, repeated code removal, and the transformation rule RJ1 to RJ6 are applied.

  - JDK: about 570k lines, in 1,877 files
  - 処理は 3 分で終了
  - Repeated code removal: 短いコードの繰り返しはクローンとみなさない

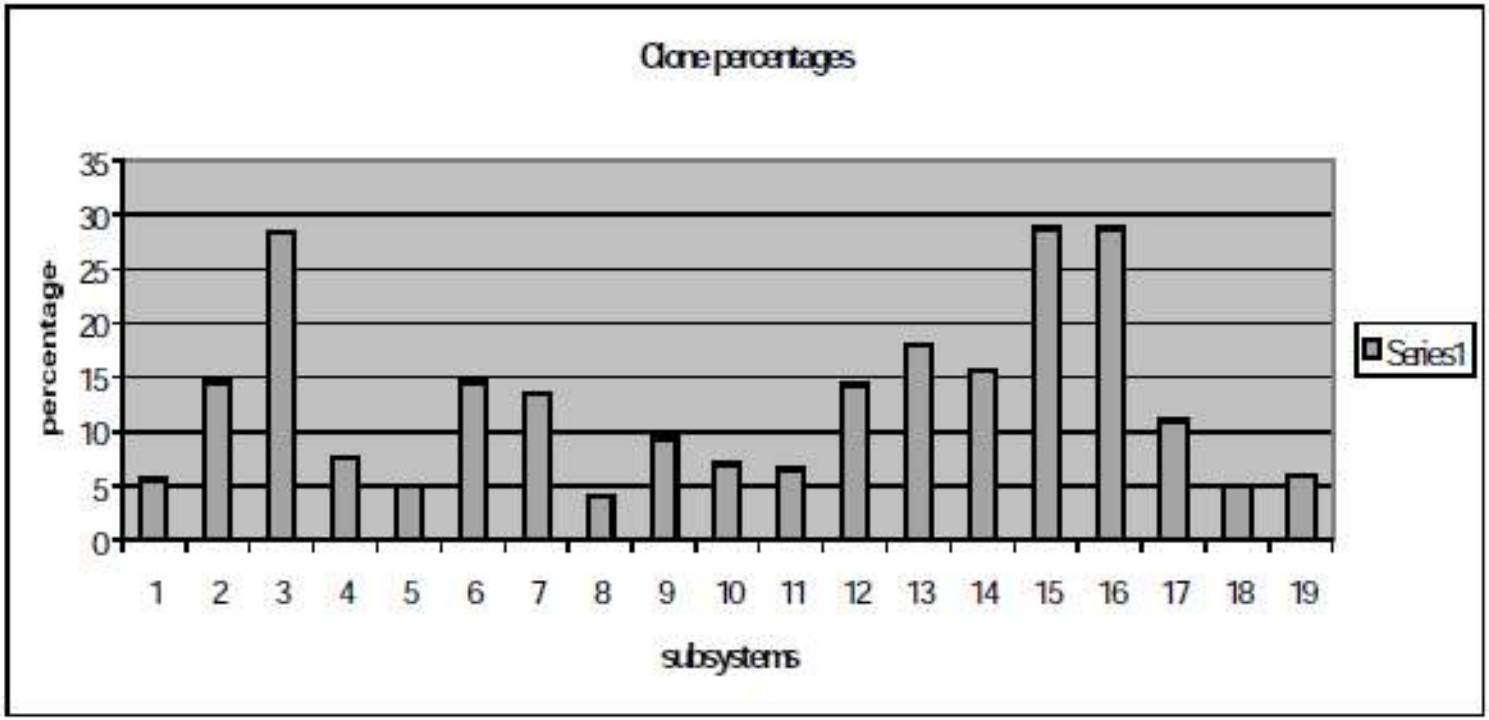
AST を用いるアプローチ [5](1/4)

- プログラムの構文木を構成
- 同じ構造の sub-trees を探す

1. 各 sub-tree をハッシュ
   - ハッシュ関数… identifier 名の違いは関知しない
2. 同じハッシュ値を持つ sub-trees のみ改めて比較
   - 探索空間を小さくするため
3. 特に sequence を表す sub-tree を優先的に探索

ASTを用いるアプローチ (2/4)

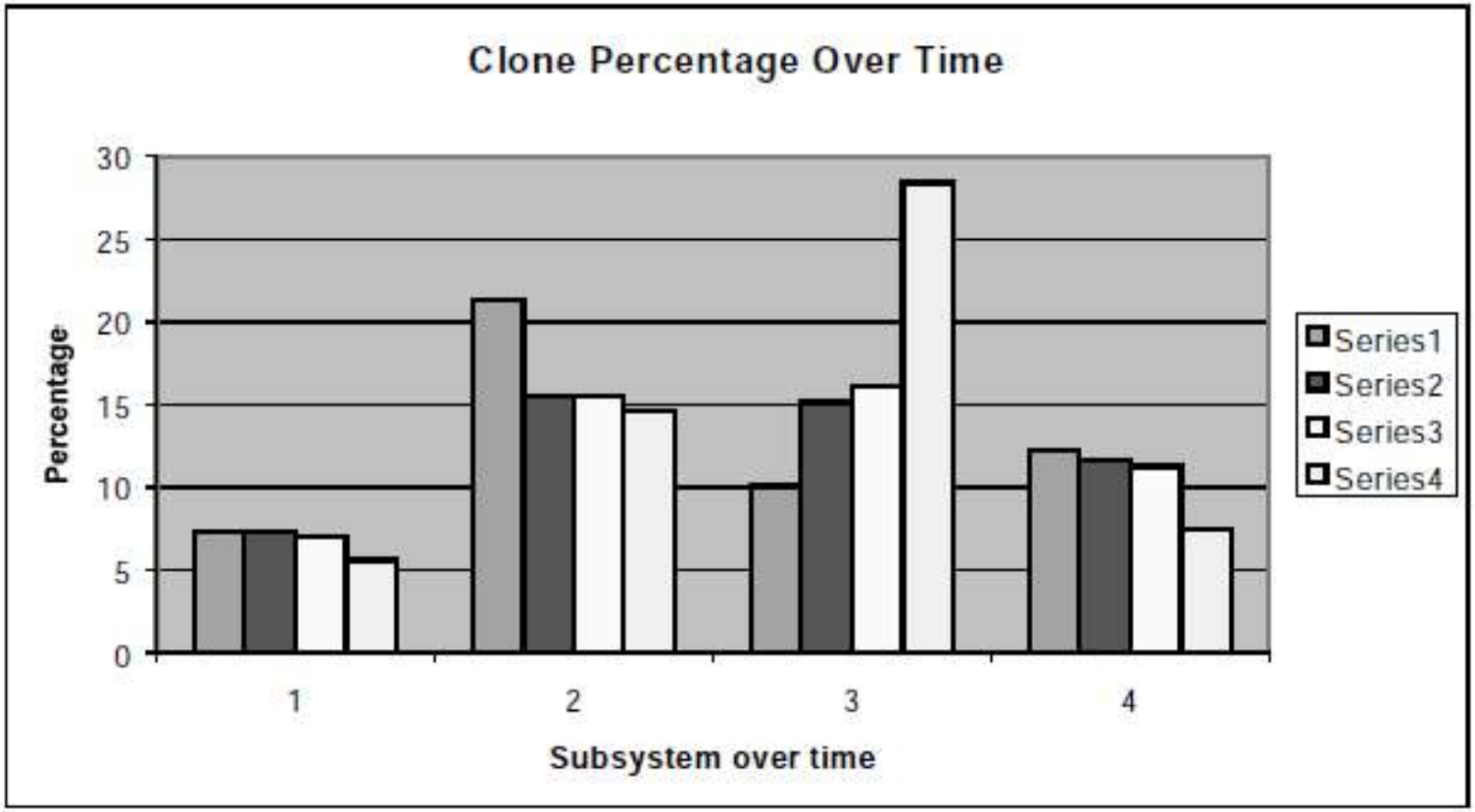
- 評価： プロセス管理システム
  - 全部で400k lines 、複数の subsystems に分かれる
  - 7 年前に作成、現在 15 人でメンテ



クローンの全体に占める割合

ASTを用いるアプローチ (3/4)

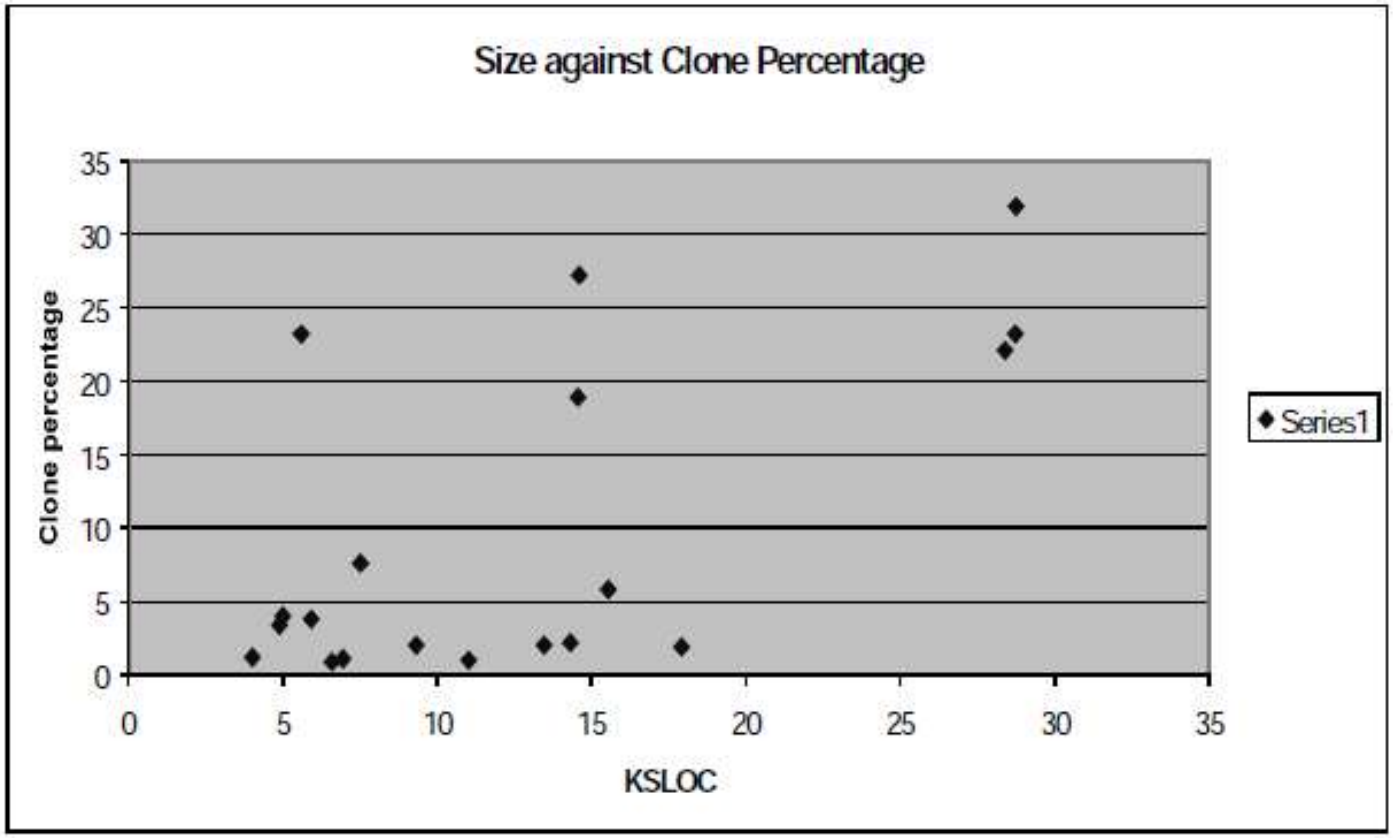
- バージョンアップによるクローン数の増減



  - おおむね減少

ASTを用いるアプローチ (4/4)

- ファイルのサイズとクローン数の関係



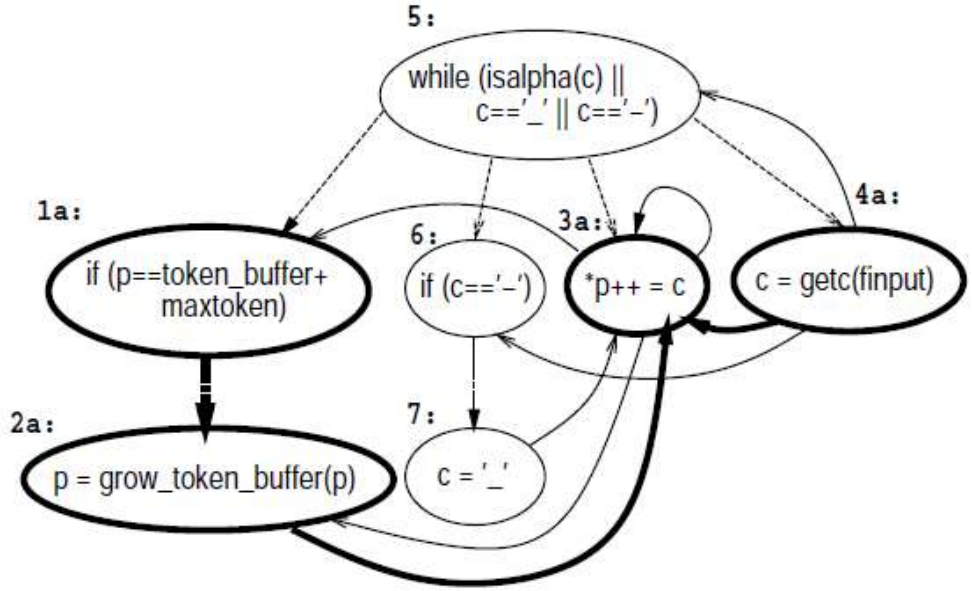
  - あまり関係ない
    - クローンの大きな発生要因は締め切りの圧力

Program Dependence Graph(1/5)

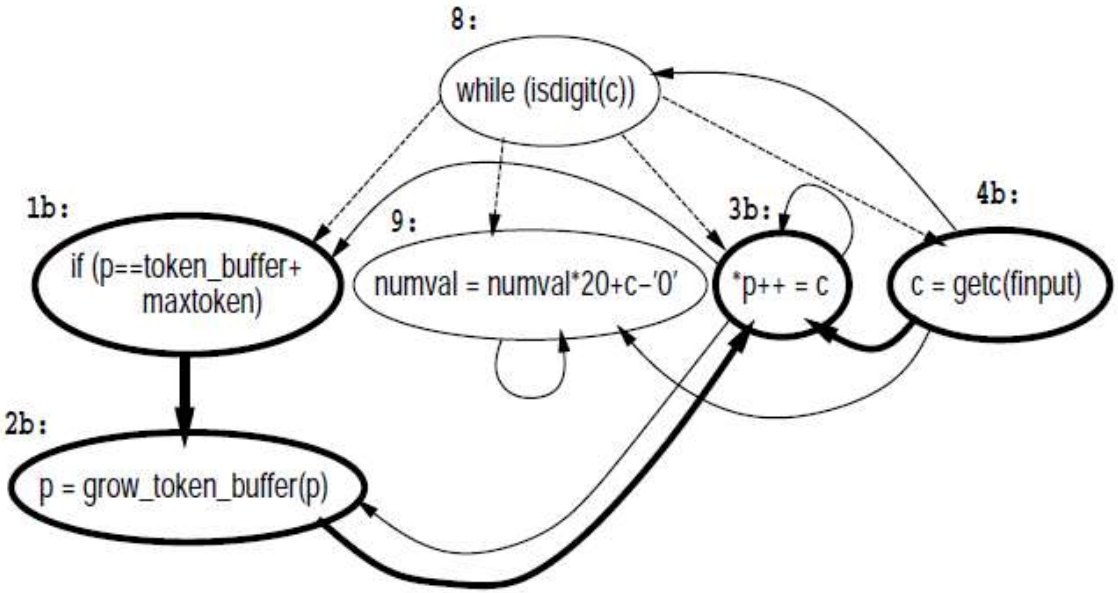
- PDG: プログラムの依存関係を表現するグラフ
  - 頂点：プログラム中の各 statement
  - 辺：データや制御の依存関係
- クローン発見問題 = 部分グラフ同型問題 [6][7]

Program Dependence Graph(2/5)

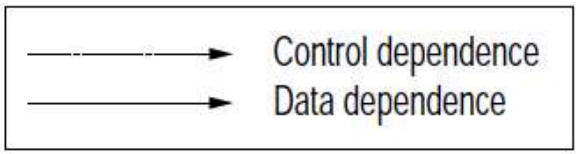
- PDG の例



PDG for Fragment 1



PDG for Fragment 2



Program Dependence Graph(3/5)

- 利点
  - 順序の入れ換え / 入り交じりに強い

```
++ tmpa = UCHAR(*a),
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++   tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx   tmpb = UCHAR(*++b);
++ if (tmpa == '-') {
     tmpa = UCHAR(*++a);
     ...
   }
xx else if (tmpb == '-') {
     if (...UCHAR(*++b)...) ...
```

Program Dependence Graph(4/5)

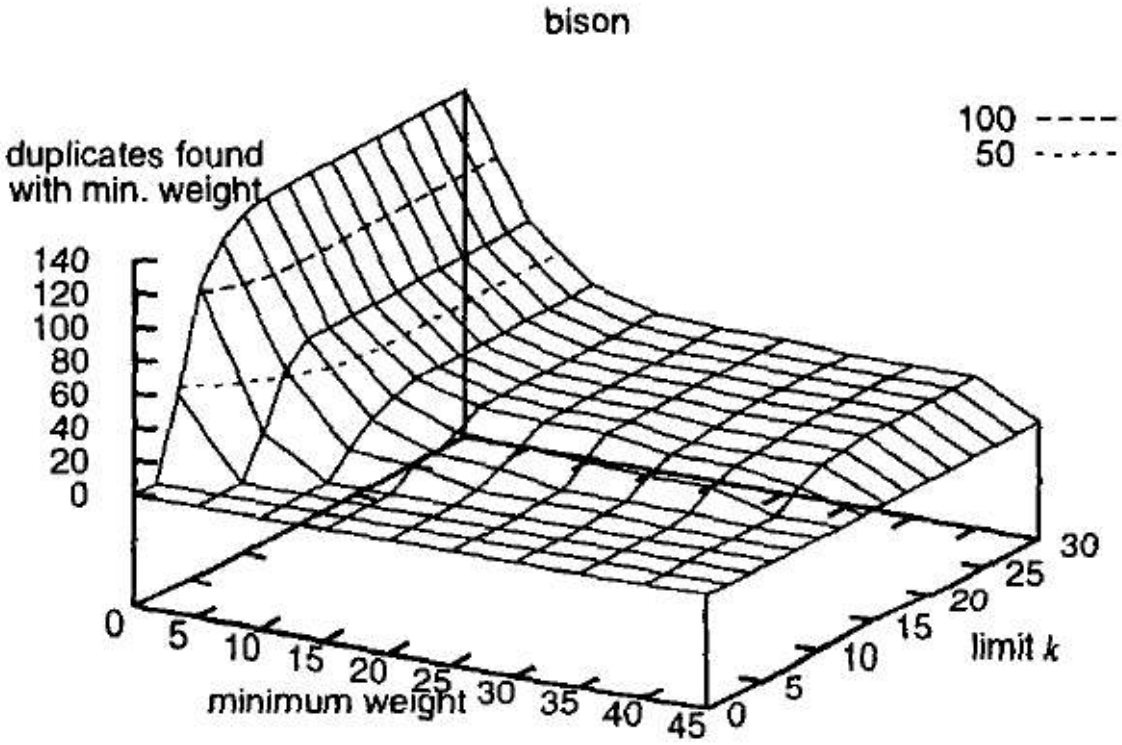
- 評価

Project	LOC	Edges	Vertices	Time f. limit k (sec)						Duplicates		
				k=10	k=20	k=30	k=40	k=50	k=100	≥10	≥20	≥50
agrep	3968	69032	22588	26.4	207.9	1465	7150	38848	-	155	91	12
bison	8303	79030	28071	8.9	47.4	249.2	714.5	920.3	921.6	34	22	0
cdecl	3879	40578	12939	0.6	0.6	0.6	0.6	0.6	0.6	0	0	0
compiler	2402	99219	16497	226.8	237.6	237.6	237.6	237.6	237.6	94	67	51
ctags	2933	45249	12446	0.6	0.8	0.8	0.8	0.8	0.8	0	0	0
diff	17485	169508	43518	2.5	9.1	32.0	61.4	63.6	63.6	40	10	0
fft	3242	35701	16446	6.0	53.4	297.2	892.9	1292	1296	16	14	8
flex	7640	124730	37073	3.3	3.8	4.2	4.3	4.3	4.3	16	0	0
football	2261	63833	18718	30.3	49.9	54.7	54.7	54.7	54.7	50	2	0
larn	10410	817432	158077	271.4	4242	5878	5905	5867	5876	91	53	6
patch	7998	196106	29766	6.27	7.5	8.6	9.2	9.3	9.2	2	0	0
rolo	5717	50816	17438	0.7	0.7	0.7	0.7	0.7	0.7	0	0	0
simulator	4476	34939	14438	1.4	2.4	2.6	2.6	2.6	2.6	0	0	0
spim	19739	1338294	122819	525.9	703.5	798.5	809.1	809.1	807.2	30	16	0
twmc	24950	1605532	181281	918.4	24263	-	-	-	-	1383	992	639

- k は検出する部分グラフのサイズ上限

Program Dependence Graph(5/5)

- よく分からないのでグラフにする



  - minimum weight は検出する部分グラフのサイズ下限
- k をある値以上に大きくしても検出数は変わらない

Higher-level Similarity Patterns(1/5)

- 共通のクローン集合を含む構造
  - 例：似ている箇所が n 箇所ある Java の Class 同士
- 単純なクローンより１つ上の概念 [8]
  - より多くのクローンを共有する Class 達を探す
    - 各ファイルに Class は 1 つだけ

Higher-level Similarity Patterns(2/5)

- 例：

CLONE PATTERN		9,9,9,15
SUPPORT		2
FILE ID	**FTC**	**FPC**
12	1175	29%
14	1175	50%

  - CLONE PATTERN: 共通に含まれるクローンの集合
  - SUPPORT: ファイルの数
  - FTC:File Token Coverage
  - FPC:File Percentage Coverage
    - CLONE PATTERN がファイルをカバーしている割合

Higher-level Similarity Patterns(3/5)

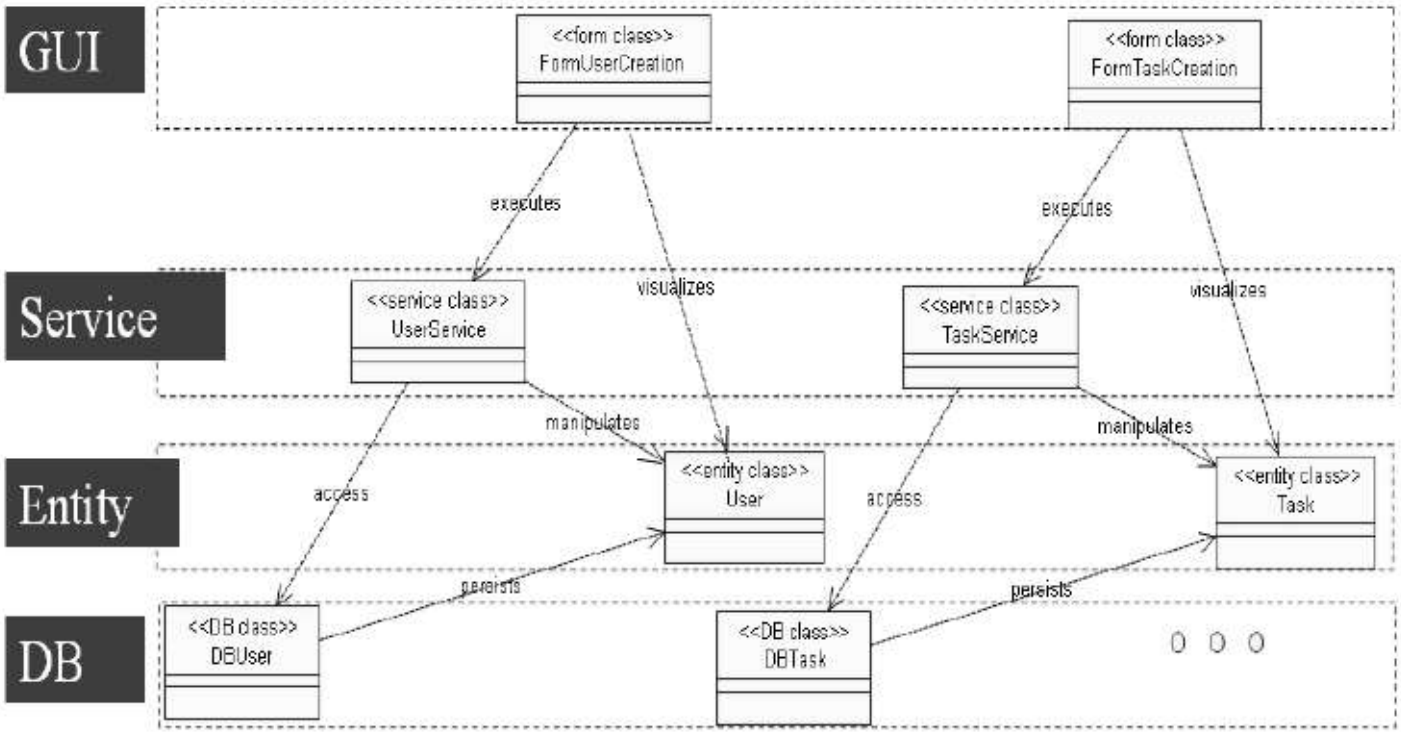
- 評価 (Java2SE1.5)
  - 類似する Class の集合を 7 個検知
    - [T]Buffer
    - Heap[T]Buffer
    - Direct[T]Buffer[S|U]
    - Heap[T]BufferR
    - Direct[T]BufferR[S|U]
    - ByteBufferAs[T]Buffer[B|L]
    - ByteBufferAs[T]BufferR[B|L]
      - T::=Byte|Char|Int|Double|Float|Long|Short
      - S は non-native な、U は native な byte ordering
      - B は big-endian 、L は little-endian

Higher-level Similarity Patterns(4/5)

- 評価 (続き)
  - 類似性がファイル名から明らかな Class を検知
    - openMBeanAttributeInfoSupport.java
    - openMBeanParameterInfoSupport.java
  - 故意に複製されたファイルも検知
    - ObjectFactory.java
      - com.sun.org.apache.* 以下に 14 個存在

Higher-level Similarity Patterns(5/5)

- 発展
  - さらに高レベルな類似性も分かるかも
    - Class の継承関係など



概要

- ソースコードの類似性を用いたさまざまな解析
  - 剽窃を検知
  - コードの複製を発見 (メインの話)
  - ソフトウェアの進化系統図を構築
  - ソースコードの作者を特定

ソフトウェアの進化系統 (1/5)

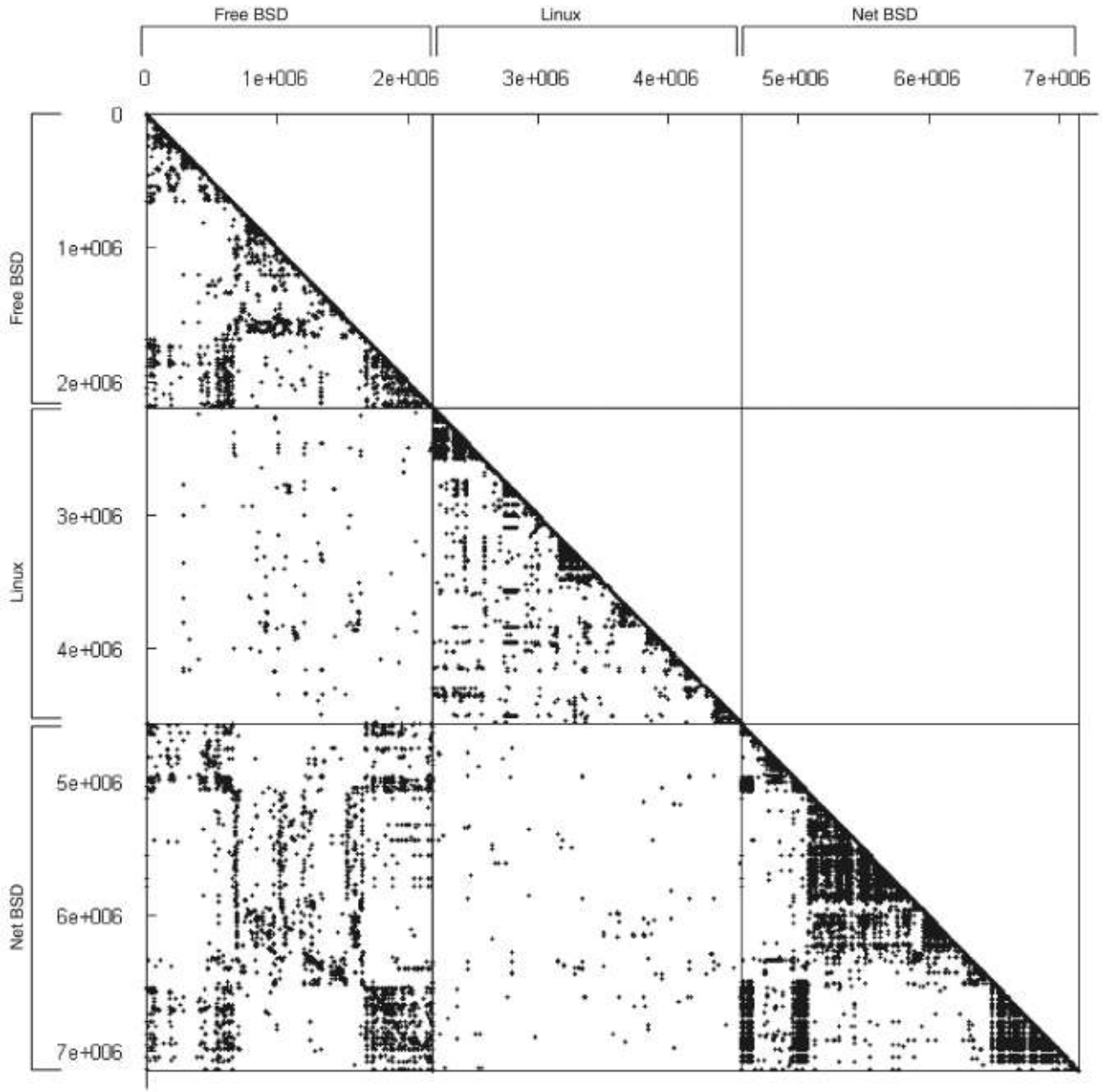
- システムの派生関係を把握
  - コードの類似性を用いて定量化可能
- 例 1: FreeBSD vs Linux vs NetBSD[4]
  - 比較行列は次頁

ソフトウェアの進化系統 (2/5)

FreeBSD

Linux

NetBSD



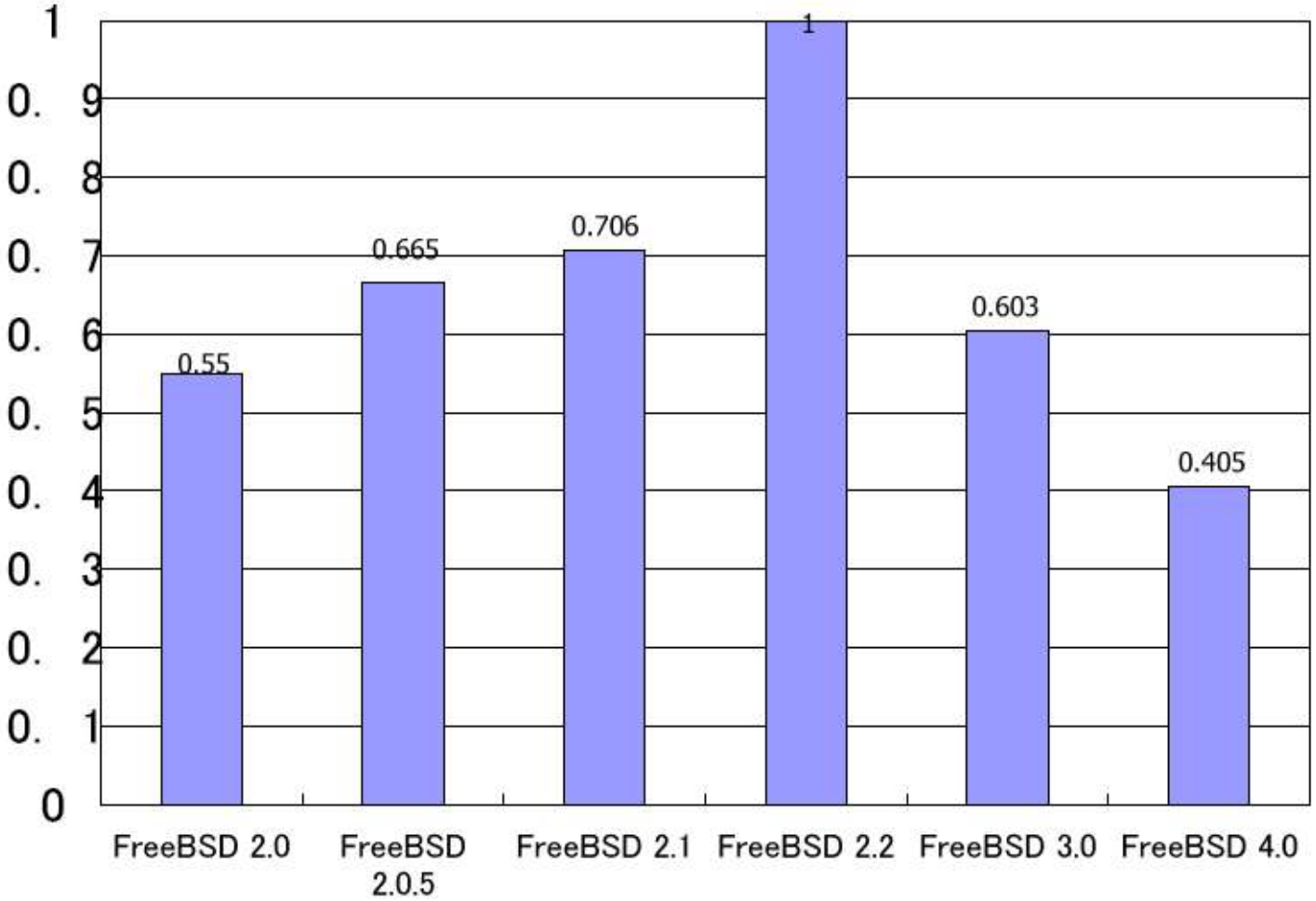
ソフトウェアの進化系統 (3/5)

- 例 2: BSD 群の系統 [9]
  - 類似度の表

	FreeBSD 2.0	FreeBSD 2.0.5	FreeBSD 2.1	FreeBSD 2.2	FreeBSD 3.0	FreeBSD 4.0	4.4BSD-Lite
FreeBSD 2.0	1.000	0.833	0.794	0.550	0.315	0.212	0.419
FreeBSD 2.0.5	0.833	1.000	0.943	0.665	0.392	0.264	0.377
FreeBSD 2.1	0.794	0.943	1.000	0.706	0.421	0.286	0.362
FreeBSD 2.2	0.550	0.665	0.706	1.000	0.603	0.405	0.226
FreeBSD 3.0	0.315	0.392	0.421	0.603	1.000	0.639	0.138
FreeBSD 4.0	0.212	0.264	0.286	0.405	0.639	1.000	0.101
4.4BSD-Lite	0.419	0.377	0.362	0.226	0.138	0.101	1.000
4.4BSD-Lite2	0.290	0.266	0.258	0.179	0.133	0.100	0.651
NetBSD 1.0	0.440	0.429	0.411	0.291	0.220	0.140	0.540
NetBSD 1.1	0.334	0.348	0.336	0.254	0.193	0.152	0.421
NetBSD 1.2	0.255	0.269	0.265	0.225	0.190	0.158	0.331
NetBSD 1.3	0.205	0.227	0.225	0.201	0.208	0.179	0.259

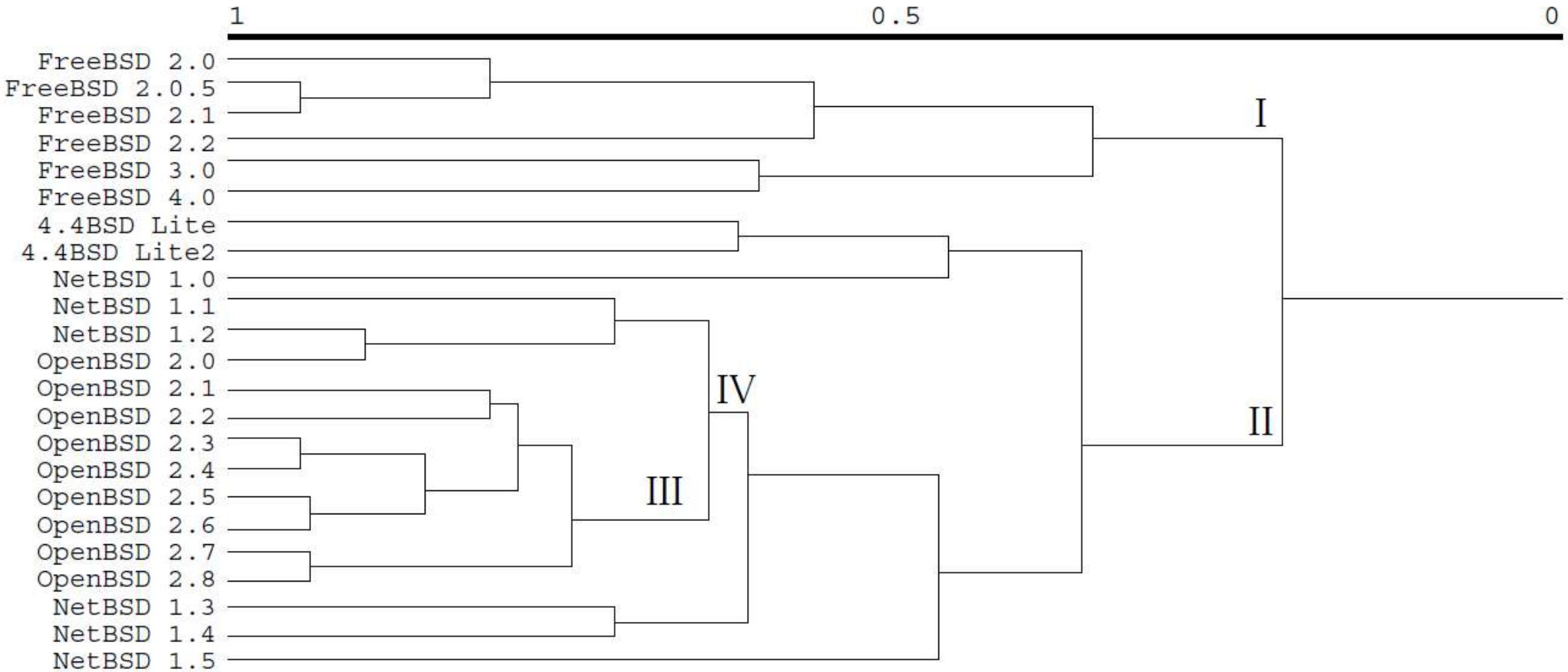
ソフトウェアの進化系統 (4/5)

- FreeBSD 2.2 と他のバージョンとの類似度
  - バージョンの近いものほど似ている



ソフトウェアの進化系統 (5/5)

- 類似度から構成したデンドログラム



- OpenBSD が NetBSD1.1 から派生した事実と合致

概要

- ソースコードの類似性を用いたさまざまな解析
  - 剽窃を検知
  - コードの複製を発見 (メインの話)
  - ソフトウェアの進化系統図を構築
  - ソースコードの作者を特定

ソースコードの作者を特定 [10](1/2)

- 同じ人が書いたコードは似ているはず
  - プログラムの類似度を用いて判定可能？
- プログラムの意味自体は重要でない
  - 剽窃検知やコードクローン検知とは違う
  - プログラミングスタイルの解析がポイント
    - インデント
    - 改行
    - コメント
    - ブロック

ソースコートの作者を特定 (2/2)

- 例

```
int strchk(char *s1)
{
  char *ptr1;

  ptr1 = s1;
  while(*ptr1 != 0)
    if(*ptr1++ == '\t')
      return(1);
  return(0);
}
```

```
#define TRUE 1
#define FALSE 0
/* Checks the existence of \t in string */
int check_for_tab_in_string(string)
char *string;
{
  char *character_pointer;

  /* Loop until found or we reach EOLN */
  for(character_pointer = string;*character_pointer != NULL;)
  {
    /* check to see if we found TAB */
    if(*(character_pointer++) == 9)
    {
      /* Success!! */
      return(TRUE);
    }
  }
  /* No tab */
  return(FALSE);
}
```

  - 剽窃かも
  - クローンかも
  - しかし、作者は同一ではない

まとめ

- ソースコードの類似性を用いたさまざまな解析
  - 剽窃を検知
  - コードの複製を発見 (メインの話)
  - ソフトウェアの進化系統図を構築
  - ソースコードの作者を特定

References(1/4)

- [1]Verco, Kristina L. and Michael J. Wise. "Plagiarism a La Mode: A Comparison of Automated Systems for Detecting Suspected Plagiarism." The Computer Journal 39.9 (1996): 741-50.
- [2]Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. Shared Information and Program Plagiarism Detection. IEEE Transactions on Information Theory, 50(7):1545--1551, 2004.
- [3]S. Ducasse, M. Rieger, and S. Demeyer. "A Language Independent Approach for Detecting Duplicated Code", Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '99, pp. 109-118. Oxford, England. Aug. 1999.

References(2/4)

- [4]Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans. Software Eng. 28(7): 654-670 (2002)
- [5]Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In ICSM, pages 368--377, 1998.
- [6]Komondoor, R. and Horwitz, S., Using slicing to identify duplication in source code. In Proceedings of the 8th International Symposium on Static Analysis, (Paris, France, July 16-18, 2001)

References(3/4)

- [7]Jens Krinke. Identifying similar code with program dependence graphs. In Proc. Eigth Working Conference on Reverse Engineering, pages 301--309, 2001.
- [8]Basit, H. A. and Jarzabek, S. 2005. Detecting higher-level similarity patterns in programs. SIGSOFT Softw. Eng. Notes 30, 5 (Sep. 2005), 156-165
- [9]T. Yamamoto, M. Matsusita, T. Kamiya, and K. Inoue. "Measuring Similarity of Large Software Systems Based on Source Code Correspondence". Technical Report of Dept. of ICS, IIP-03-03-02, 2002.

References(4/4)

- [10]I. Krsul. "Authorship analysis: Identifying the author of a program". Technical report, Department of Computer Science, Purdue University, 1994. Technical Report CSD-TR-94-030.