米澤研究室全体ミーティング
(2005/11/08)

M1　佐藤秀明

概要

- コードの複製を検知 / 処理する方法
  - 既存研究
  - 自分の研究方針

Code Clone

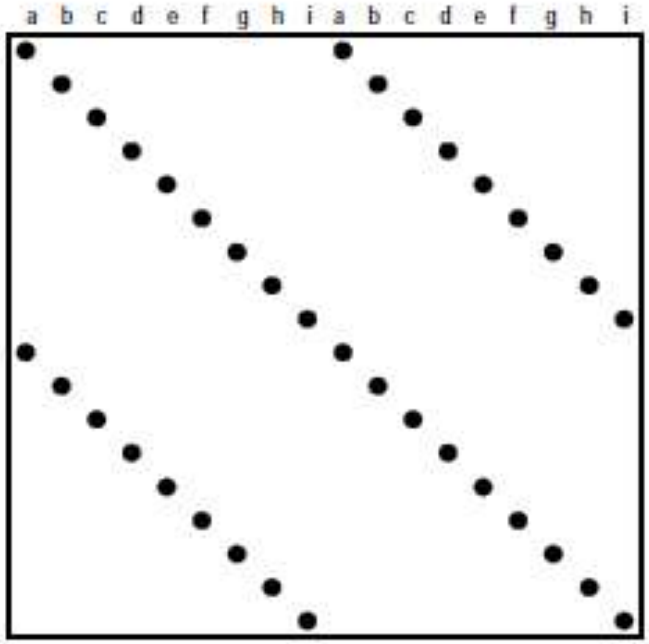
- 機能または文面が似ているコード
  - ソースをコピー & ペースト
  - 設計の洗練不足
- 大規模なシステム開発において有害
  - メンテナンス性の低下
- ソースコードの類似性を解析して発見
  - 「類似」の定義は？

Code Clone の検知方法

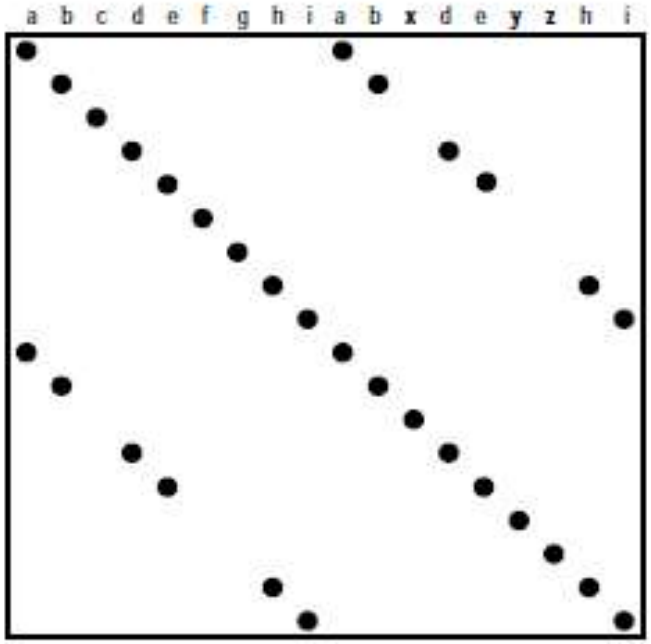
- 扱うデータ構造
  - 行単位での解析
  - トークン単位での解析
  - 構文木単位での解析
  - Program dependence graph の利用
- 用いるアルゴリズム
  - 完全マッチ
  - parameterized suffix tree(p-suffix tree)
  - ハッシュ
  - グラフ同型問題

言語非依存な手法 [Ducasse ら](1/3)

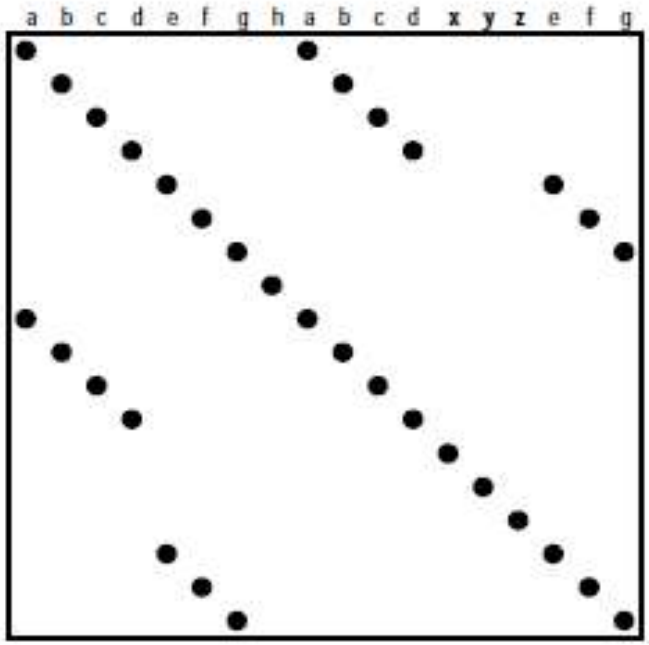
- 各行ごとの完全マッチで評価
  - 処理が軽い
  - さまざまな言語に適用可能
- 比較行列の作成
  - マッチした部分を視覚的に表現
    - row と column はファイルの行ナンバー
    - マッチングに対応する成分の値は 'true'
  - 次頁は比較行列の例



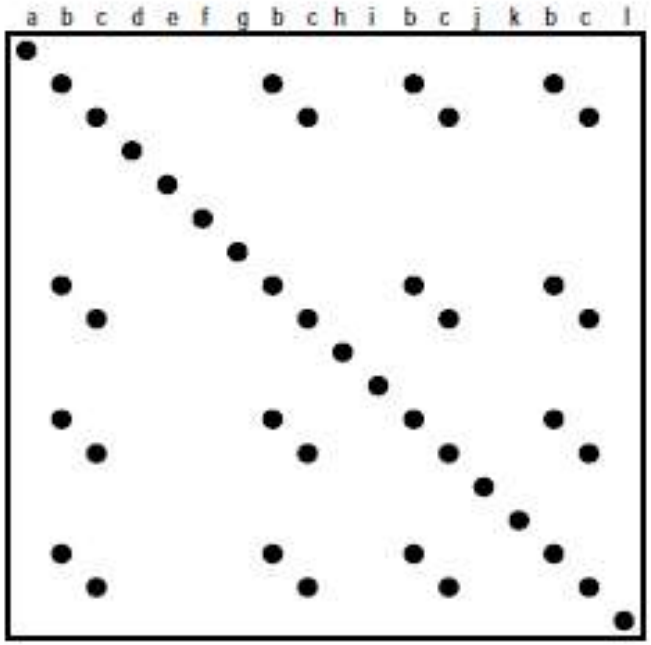
完全に一致



一部変更あり



一部追加 / 削除あり



周期的な出現

言語非依存な手法 (3/3)

- 問題 1: 遅い
  - gcc(460000 行) の解析に 6 時間
  - 実装 / 実験環境の問題か
- 問題 2: 行単位では粒度大きすぎ
  - 変数のリネームに対応できない

CCFinder[Kamiya ら](1/3)

- トークン列のマッチング
  - p-suffix tree
- 字句解析後に種々の変換を実行
  - 言語依存の変換ルール
    - 名前空間 / アクセス属性などを削除
    - プログラム構造の明確化
  - 変数名の置換
    - 変数は同一の special case に置換
- プログラマに判断材料を提示
  - そのクローンを除去すると行数がどれだけ減るか
  - そのクローンはどれくらい散らばって存在しているか

CCFinder(2/3)

- Suffix tree
  - トークン列の全接尾語を保持する trie
- Parameterized-suffix tree[Baker]
  - 同じ変数の出現を直前の出現位置からの距離に置換
    - 例 :xbyyxbx ⇒ 0b014b2
  - マッチングに使うと速い
    - JDK1.3.0(570000 行) を 3 分

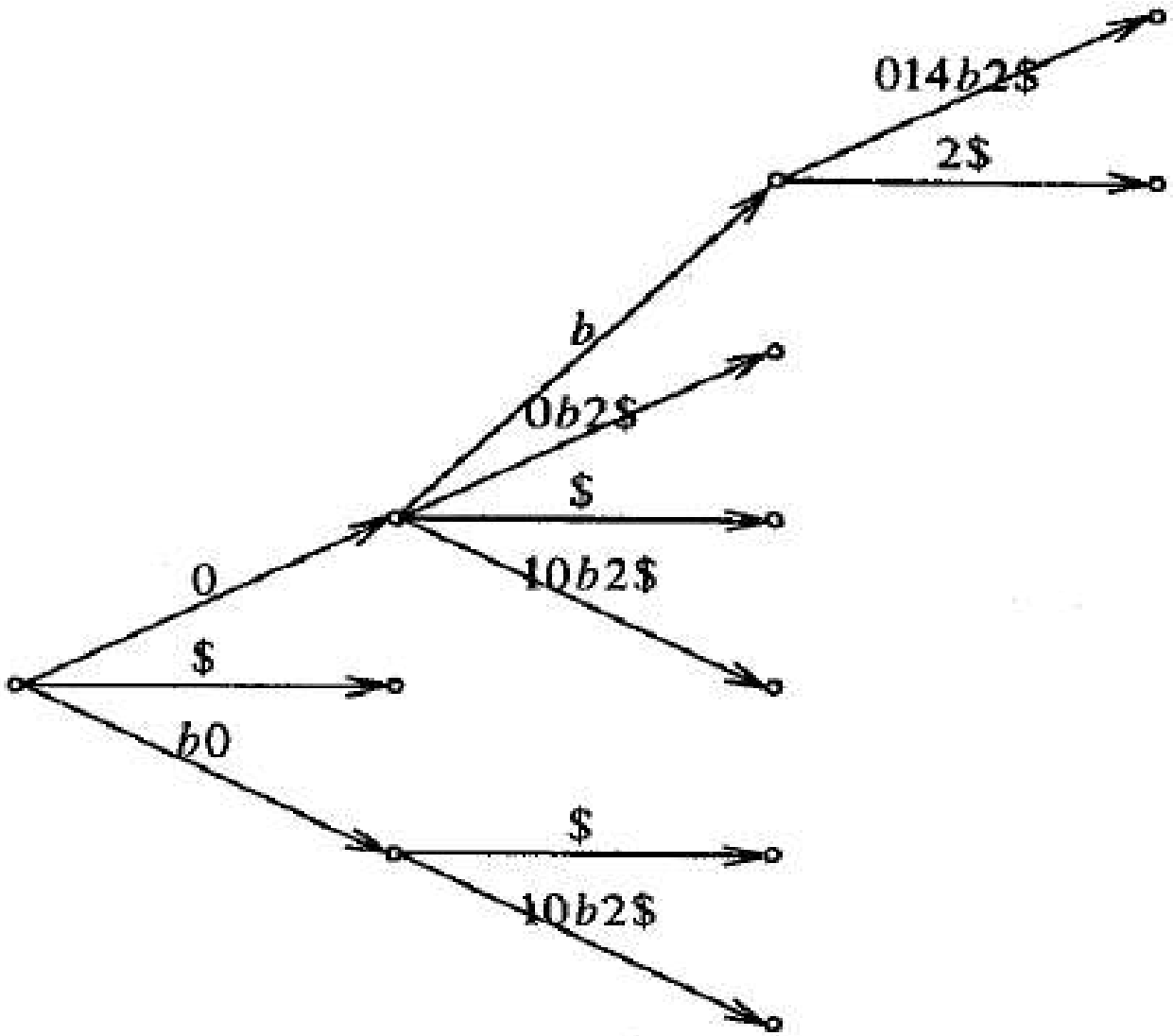


Figure 5: A p-suffix tree for the p-string $S = xbyyxbx\$$.

構文解析による手法 [Baxter ら]

- プログラムの構文木を構成
- 同じ構造の sub-trees を探す

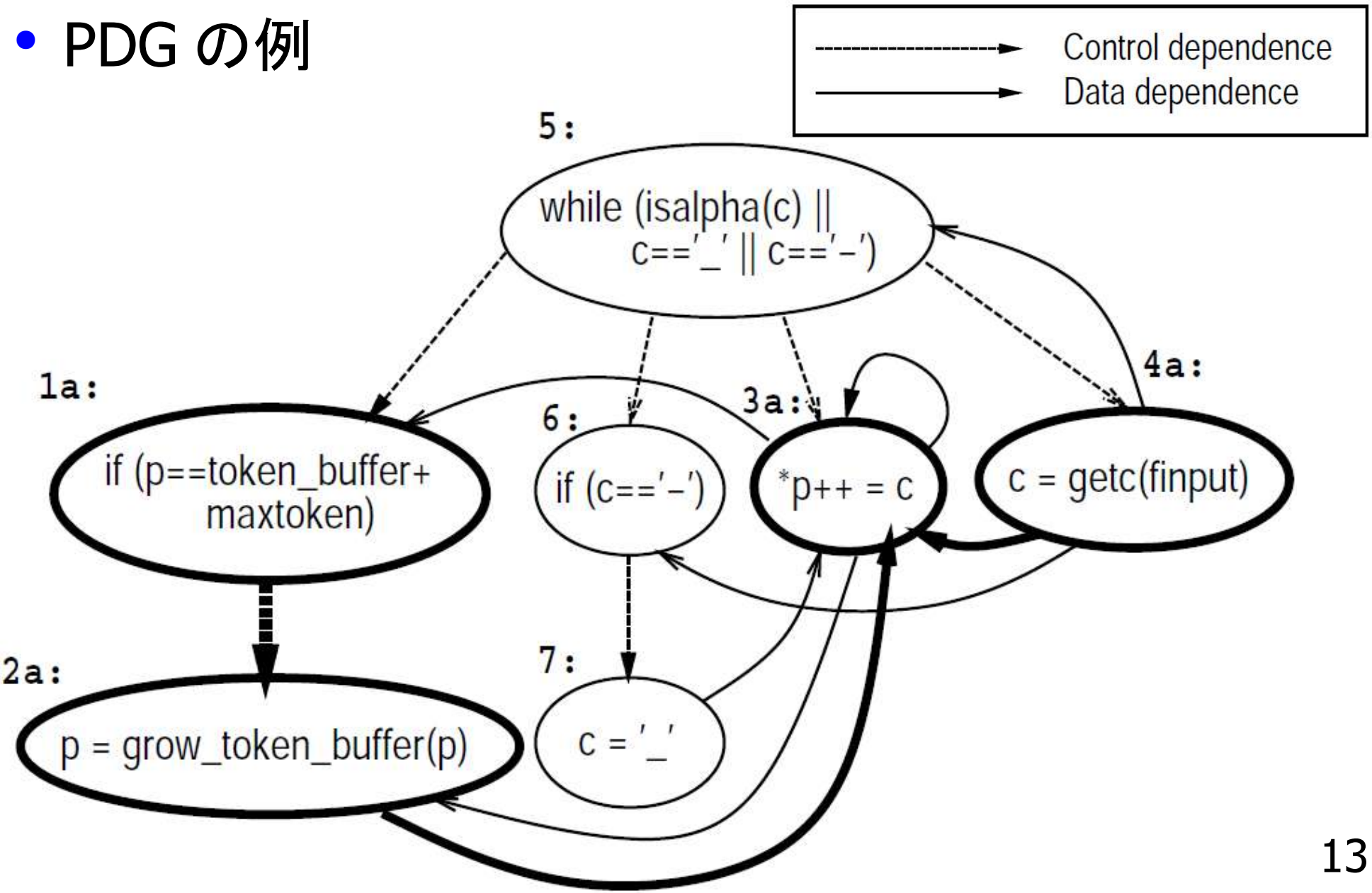
1. 各 sub-tree をハッシュ
   - ハッシュ関数… identifier 名の違いは関知しない
2. 同じハッシュ値を持つ sub-trees のみ改めて比較
   - 探索空間を小さくするため

Program Dependence Graph(1/3)

- 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 [Komondoor ら][Krinke]
  - 利点：他のコードが入り組んでいるクローンも検知可能

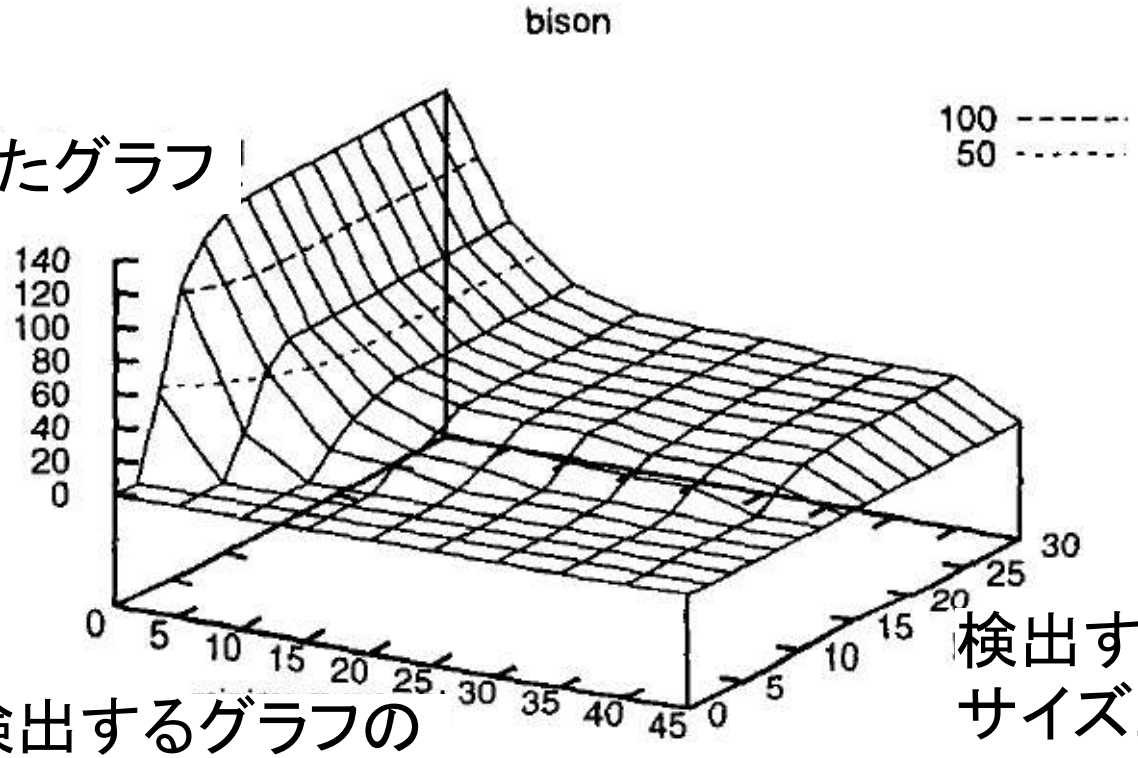
Program Dependence Graph(2/3)

- PDG の例



Program Dependence Graph(3/3)

- 評価



検出したグラフ

検出するグラフの
サイズ下限

検出するグラフの
サイズ上限

- サイズ上限を増やしても結果が変わらなくなる閾値の存在

検知から除去へ

- クローンを検知 / 分析する手法は昔からある
- クローンを除去する手法の既存研究は少ない
  - マクロにまとめる [Baxter ら]
  - デザインパターン [Balazinska ら]
- 技術的な困難はどこにあるのか？

自動除去が困難な要因 (1/2)

- 最低でも構文解析まで行う必要性
  - 共通なコード片を分離する操作

- 膨大な計算量
  - 時間的 : 構文木の作成 / マッチングは時間がかかる
  - 空間的 : 木構造がメモリを独占
    - 実行時間やメモリ量を評価している論文はない

自動除去が困難な要因 (2/2)

- クローンを除去するかどうかは人間が判断
  - そのときの状況に依存
    - プログラム構成の保存を重視する場合
    - メンテナンス性より実行効率を重視する場合
  - 人間の立ち合いが必要
    - 自動で判断させるのは難しい

- 人間を何時間も待たせるような処理ではだめ
  - できるだけ早く応答したい

解決の糸口 (1/2)

- クローン候補を 2 段階で選抜
  - 第 1 段階 : 負荷の小さい解析
  - 第 2 段階 : 負荷の大きい解析
    - 簡易な解析で候補をあらかじめ絞っておく
- 実行時間とメモリ使用量の削減をはかる

解決の糸口 (2/2)

- 問題を分割し、解けたものから次々に提示
  - ユーザの処理と他の仕事を同時に走らせる
- トータルの所要時間を削減
  - ユーザが判断を下すまでの時間を有効に利用

問題設定 (1/2)

- 対象言語 :C 言語
  - 評価を行うにふさわしいシステムが豊富
    - 大きなシステム
    - 長期にわたって開発 / 保守されているシステム
- 変数のリネームは関知
  - リネーム以上に改変されたコードは検知しない
    - マッチング判定はもう少し緩めるかも
  - コピー & ペーストされたものが対象

問題設定 (2/2)

- クローン解消の方法は関数抽出のみ
  - 変数のリネームくらいであればこれで十分
  - それ以上は言語自体がもっとリッチでないと無理か
    - オブジェクト
    - 関数のオーバーロード
    - 高階関数
- 3 個以上のコード片もクローンとして検知
  - 2者のマッチングを報告するだけでは不十分

クローン候補の選抜

- 第 1 段階 : トークン単位の解析
  - parameterized suffix tree の relax 版
    - 細々とした差異は吸収したい
- 第 2 段階 : 構文木単位の解析
  - パーズ
  - 型チェック
  - 生存期間解析
  - 自由変数解析

関数抽出

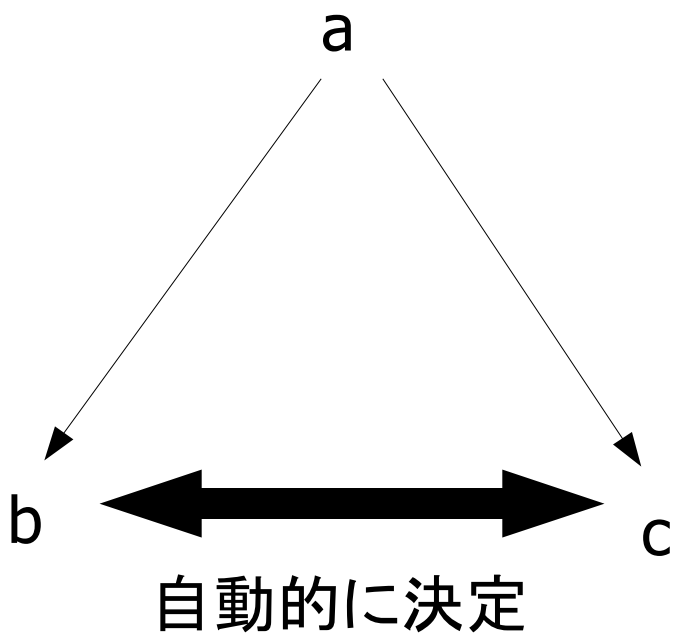
- 共通のコード片を関数としてくくり出す
  - 自由変数は関数の引数に追加
  - 値の変化する変数を返り値にする
    - 2 つ以上あるときはポインタで渡す？

同値関係の導入(1/2)

- コード片 a が b にマッチするという関係 (∽) を定義
  - a ∽ a
  - a ∽ b⇒b ∽ a
  - a ∽ b∧b ∽ c⇒a ∽ c
- 互いの一部がオーバラップしたコード片は別物とする
  - クローン同士ののコンフリクトについては後述

同値関係の導入 (2/2)

- 異なる 2 人の友人は、互いに友人関係にある
  - 同値関係より $a \backsim b \wedge a \backsim c \Rightarrow b \backsim c$
- 自分とマッチする複数のコード片は全て互いにマッチするとみなす
  - $b \backsim c$ は確認しなくてもよい



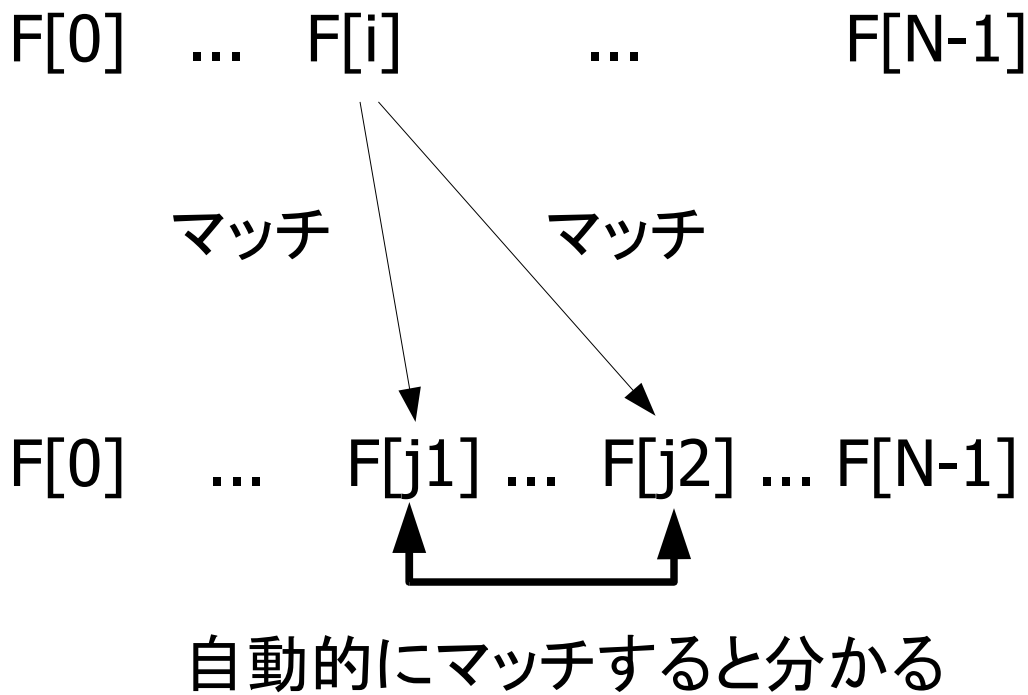
問題分割のアルゴリズム (1/2)

- 全ソースコードを N 個の要素に分割
  - F[0] から F[N-1] まで
  - 基本的にファイル単位で分割
- ２要素間の類似部分を総当たりで検知
  - 2 重ループ
  - クローンが確定する度にすぐユーザへ報告、判断を仰ぐ

```
for(i = 0; i < N-1; i++){
    for(j = i; j < N-1; j++){
        F[i] と F[j] の類似部分を検知
    }
}
```

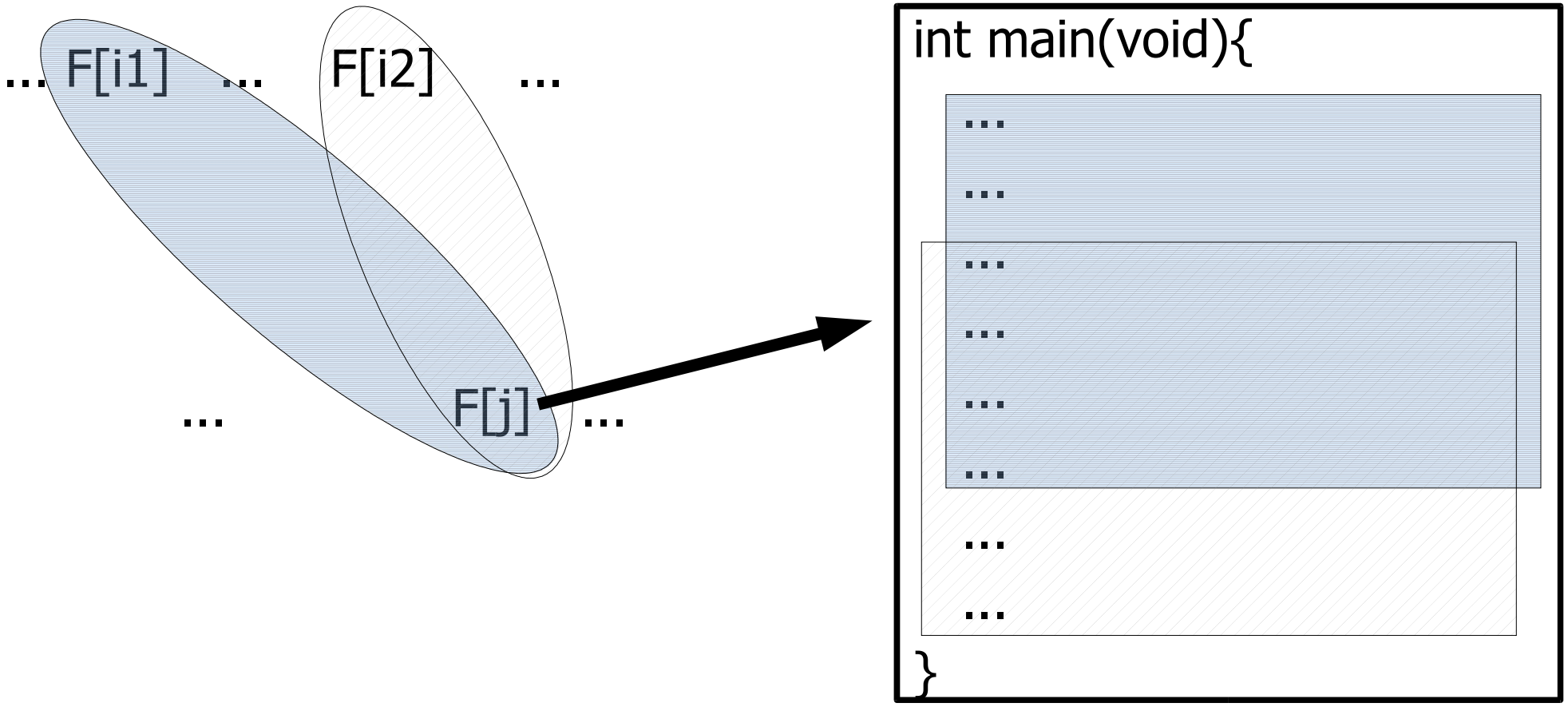
問題分割のアルゴリズム (2/2)

- 内側ループを抜けたら F[i] の関与するクローンは確定
  - 同値関係より、$i \backsim j \wedge i \backsim k \Rightarrow j \backsim k$
  - 全体にかかる時間の約 1/n 倍で最初のクローンが確定



クローンのオーバラップ (1/2)

- 異なるクローンが少しだけずれて重なる場合



クローンのオーバラップ (2/2)

- 変換後のソースコードはキャッシュする
  - 全ての解析は変換前のソースについて実行
  - 最後に write-back
- 異なるクローンがダブりそうになったらすぐ報告
  - 該当コード片をどちらのクローンに所属させるかユーザが選択
  - ある token が所属できるのは高々 1 個のクローン
  - GUI で整理して状況提示するといいかも

より攻撃的なクローン報告

- マッチングのペアが分かった時点ですぐ報告
  - 内側ループの終了を待たない
  - あとから判明した同クローンに属するコード片は追々報告
    - その都度処置方法を聞く or あらかじめ処置を決めておく
- 途中でわけが分からなくなりそう
  - GUI 必須

キャッシュサイズとの兼ね合い

- 分割した小問題がちょうどキャッシュに乗れば速そう
  - ソースコードのサイズ
  - suffix tree のサイズ
  - 構文木のサイズ
- ソースコードをファイル単位ではなく柔軟に分割
  - どの処理フェーズが重いか分かってから考えることにする
    - たぶん構文木の処理がいちばん重い

並列計算

- 構文木はメモリを食う
  - JRE の全ての構文木は 3GB に収まらない
- クラスタ等を使うと効率よく解けそう
  - 既に問題分割しているので適用しやすい
  - MPI 等を使うことになるか
  - だんだん自然言語処理みたいになってきた

全部自動で行うためには

- false positive と false negative のさらなる削減
  - 人間にとって有用なクローン候補とは何か？
- ユーザの設計ポリシーを綿密に指定できる枠組み
  - どのようなクローンを除去したいか

実装

- とりあえず OCaml で実装
  - lex
  - parse
  - 種々のプログラム解析 (CIL framework)
- 他のシステムとの連携
  - GUI を作るなら X か Java
  - 並列をやるなら MPI

まとめ

- コードの複製を検知 / 除去するしくみ
  - 検知に関する既存研究
  - 除去に関する自分の研究方針
    - 2 段階選抜による計算量の削減
    - 問題分割による迅速な報告