Mining Patterns in Source Code using Tree Mining Algorithms

Hoang Son Pham, Siegfried Nijssen, Kim Mens,
Dario Di Nucci, Tim Molderez, Coen De Roover,
Johan Fabry, and Vadim Zaytsev

22nd International Conference on Discovery Science
SPLIT, CROATIA, OCTOBER 28-30, 2019

Outline

Introduction

Frequent tree mining

Constraint-based maximal tree mining

Empirical evaluation

Conclusion and future work

Introduction

Discovering source code patterns: coding idioms, usage protocols

Source code patterns can be used to build intelligent systems to support maintaining and developing big software projects

```
public final class ReturnCountCheck extends AbstractFormatCheck {
   ...
   @Overwrite
   public int[] getDefaultTokens(){
      return new int[] {
         TokenTypes.CTOR_DEF,
         TokenTypes.METHOD_DEF,
         TokenTypes.LITERAL_RETURN,
         };
      }
   ...
   @Overwrite
   public int[] getRequiredTokens() { … }
   ...
   @Overwrite
   public void visitToken(DetailAST aAST){
      switch (aAST.getType()) {
         case TokenTypes.CTOR_DEF:
         case TokenTypes.METHOD_DEF:
         ...
         break;
         ...
         default:
         ...
         }
   }
   ...
   @Overwrite
   public void leaveToken(DetailAST aAST){
      switch (aAST.getType()) {
         case TokenTypes.CTOR_DEF:
         case TokenTypes.METHOD_DEF:
         ...
         default:
         ...
         }
   }
   ...
}
```

Example of pattern in source code

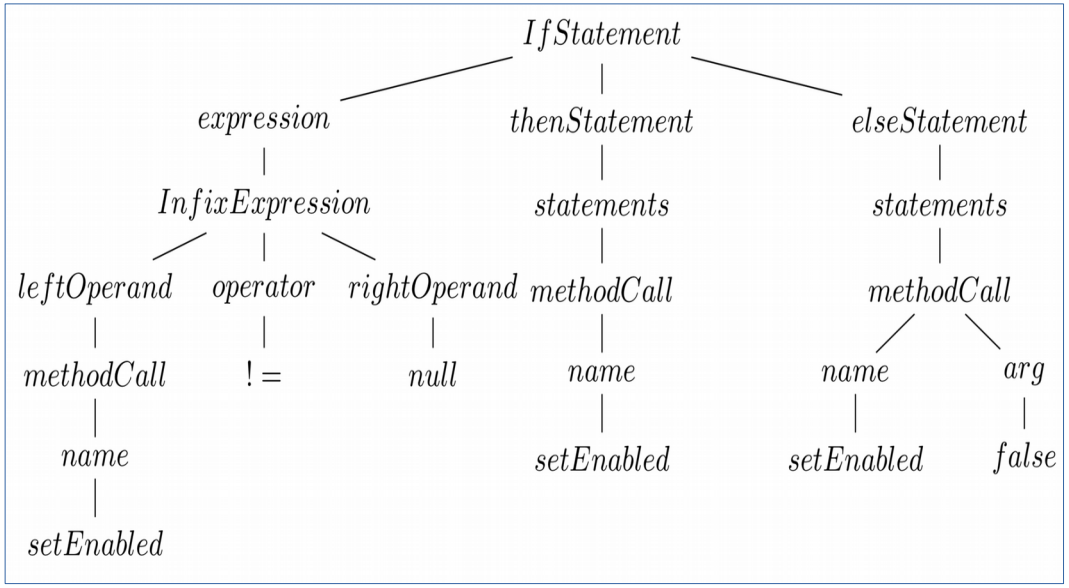
Source code patterns

- Source code are represented by Abstract Syntax Trees (ASTs)
- Source code patterns correspond to fragments that occur frequently
- Discover source code patterns: frequent tree mining algorithms

Source code pattern

```
protected void updateEnabledState() {
    if (getView() != null) {
        setEnabled(...);
    } else {
        setEnabled(false);
    }
}
```

Tree representation



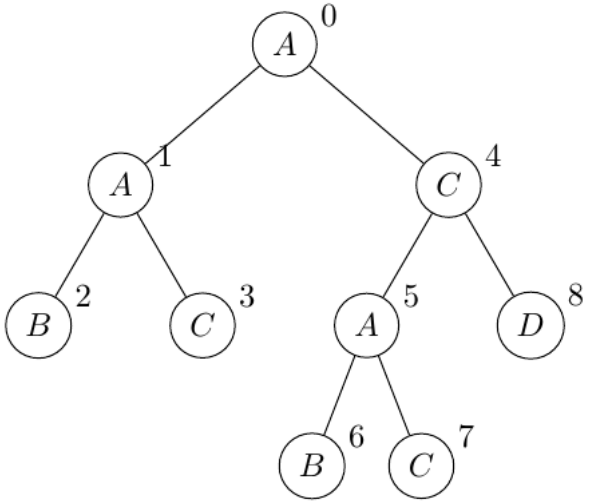
Frequent tree mining

Ordered, rooted tree mining

- FREQT, a depth-first search tree mining algorithm
- Limitation: finds large numbers of patterns

Other approaches:

- Constraint-based pattern mining
- Condensed representations, e.g., closed, maximal frequent tree mining
- Statistically motivated pattern set mining
- → *not designed to work on ASTs*



Example of ordered, rooted tree

Our proposal

Novel constraint-based tree mining algorithm, specifically designed for the analysis of code repositories

Our approach based on FREQT with these extensions:

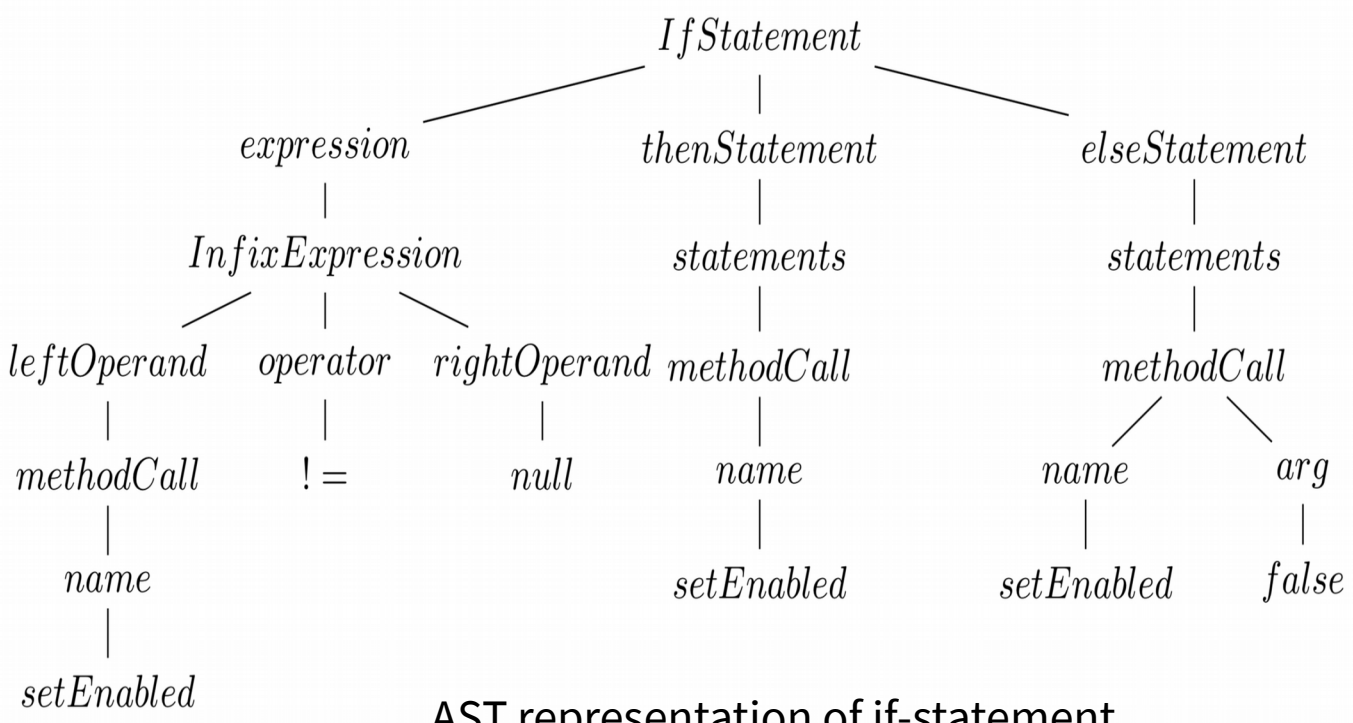
(i) **constraint-based data mining**, in which additional constraints are imposed on the patterns to be found

(ii) **maximal frequent subtree mining** to ensure that a condensed representation of only large patterns is found

Interesting source code patterns

Reflect structure of programming language

Total number of nodes/leafs in a pattern sufficiently large



AST representation of if-statement

Proposed constraints

***C0*: minimum support**

***C1*: maximum size of the pattern**

***C2*: minimum size of the pattern**

***C3*: limit the set of labels allowed to occur in the root of patterns**

***C4*: provide labels forbidden from occurring in the pattern**

***C5*: limit the number of siblings in a pattern that can have the same label**

***C6*: all leaf nodes in a pattern must have a label that is included in leafs of data**

***C7*: do not allow to produce patterns that missing mandatory labels**

Constraint-based maximal tree mining algorithm

FREQTALS algorithm

input: tree data $\mathcal{D}$, constraints C_0-C_5

output: a set of maximal patterns

Step 1: using modified $\mathcal{FREQT}$ to find frequent patterns under the given constraints

Step 2: growing the patterns found under constraint as large as possible, and return the maximal patterns among these large patterns

Empirical evaluation

Dataset:

- Programming language: Java
- Project: CheckStyle (including 270 classes)

Configurations:

Constraint	Variable	Value
C0	Minimum Support Threshold	5
C1	Maximum # of Leaves	4
C2	Minimum # of Leaves	2
C3	Root Labels	TypeDeclaration, Block
C4	Black List Labels	Javadoc, Modifiers, Annotations, ...
C5	Maximum # of Similar Siblings	10

Qualitative Analysis

```
public final class ReturnCountCheck extends AbstractFormatCheck {
   ...
   @Overwrite
   public int[] getDefaultTokens(){
      return new int[] {
         TokenTypes.CTOR_DEF,
         TokenTypes.METHOD_DEF,
         TokenTypes.LITERAL_RETURN,
         };
      }
   ...
   @Overwrite
   public int[] getRequiredTokens() { … }
   ...
   @Overwrite
   public void visitToken(DetailAST aAST){
      switch (aAST.getType()) {
         case TokenTypes.CTOR_DEF:
         case TokenTypes.METHOD_DEF:
         ...
         break;
         ...
         default:
         ...
         }
   }
   ...
   @Overwrite
   public void leaveToken(DetailAST aAST){
      switch (aAST.getType()) {
         case TokenTypes.CTOR_DEF:
         case TokenTypes.METHOD_DEF:
         ...
         default:
         ...
         }
   }
   ...
}
```

Pattern 34: An instance of Checkstyle's Visitor design pattern

```
private void visitMethod(final DetailAST aMethod)
   ...
   DetailAST child = objBlock.getFirstChild();
   while (child != null) {
      if (child.getType() == TokenTypes.METHOD_DEF) {
         ... }
      child = child.getNextSibling();
}
```

Pattern 27: AST traversal.

```
@Override
public void leaveToken(DetailAST aAST) {
   switch(aAST.getType()) {
      case TokenTypes.OBJBLOCK:
      case TokenTypes.SLIST:
      case TokenTypes.LITERAL_FOR:
      ...
}
```

Pattern 9: Check for different blocks.

```
public int[] getDefaultTokens(){
    return new int[] {
       TokenTypes.ASSIGN,            // '='
       TokenTypes.DIV_ASSIGN,        // "/="
       TokenTypes.PLUS_ASSIGN,       // "+="
       …
    };
}
```

Pattern 18: Method structure.

```
private boolean checkParams(DetailAST aMethod){
   ...
   if ((aAST.getType() == TokenTypes.VARIABLE_DEF) ||
       (aAST.getType() == TokenTypes.PARAMETER_DEF))
      {
        …
      }
```

Pattern 140: IF Statement.

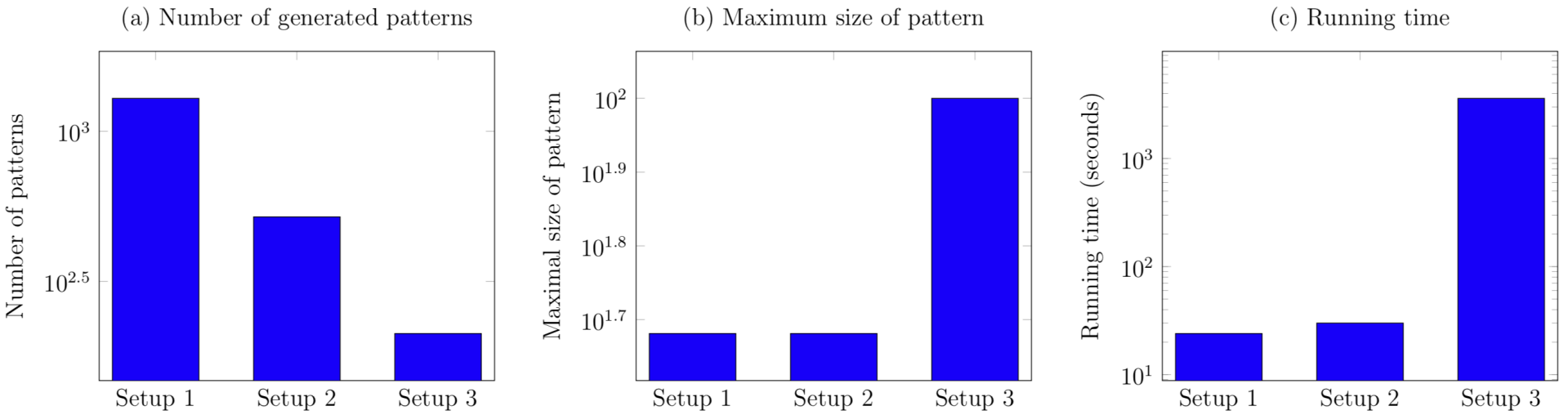
Quantitative analysis

1. Compare to the original FREQT algorithm

   FREQTALS : 1,288 patterns in 23 seconds

   Original FREQT : 717,859 patterns in 60 minutes

2. Evaluate the extensions of FREQTALS



Conclusion and Future work

Conclusion:

- discovered patterns highlight relevant code regularities
- patterns found are significantly larger
- amount of output patterns is significantly smaller
- configurations are easily adapted to other programming languages

Future work:

- employ FREQTALS on legacy programming languages, e.g., COBOL
- explore more details of the quality of patterns
- improve the scalability
- define guidelines to setup configurations

Thank for your attention!