Finding similarities through factorization

Michel Chilowicz Étienne Duris Gilles Roussel
Email: firstname.lastname@univ-paris-est.fr
Website: http://igm.univ-mlv.fr/~chilowi/finding_similarities/

Laboratoire d'Informatique de l'Institut Gaspard-Monge, CNRS
Université Paris-Est

8th workshop on Language Description, Tools and Applications
Budapest
Saturday April 5th 2008

Outline

Contexts of source code similarities detection

Studied contexts

- Comparing a pair of projects (e.g. commercial software plagiarism cases)
- Detecting similarities inside a cluster of projects (e.g. student projects)

Contexts of source code similarities detection

Studied contexts

- Comparing a pair of projects (e.g. commercial software plagiarism cases)
- Detecting similarities inside a cluster of projects (e.g. student projects)

Other contexts

- Finding redundancies into a project to refactor it
  - Exact pattern matching
  - Comparison of abstract syntax trees [Baxter, 1998]
  - Comparison of program dependence graphs [Krinke, 2001]
- Searching chunks of code from a project against a database of projects
  - Vectorial code metrics (Halstead complexity measures, ...)
  - Source code fingerprinting (Karp-Rabin hashing, Winnowing fingerprint selection, ...)

Detecting similarities inside a cluster of k projects

Comparing the $O(k^2)$ pairs of projects from the cluster

Classical string-pair comparison algorithms on tokenized projects of mean size n (inspired from genomics):

- Global alignment with dynamic programming ($O(n^2)$)
- Local alignment with dynamic programming [Irving, 2004] ($O(n^2)$)
- Extending matching parts via germs fingerprinting (in $O(n)$): Greedy String Tiling

Drawback: global temporal complexity between $O(k^2 n)$ and $O(k^2 n^2)$

Detecting similarities inside a cluster of k projects

Comparing the $O(k^2)$ pairs of projects from the cluster

Classical string-pair comparison algorithms on tokenized projects of mean size n (inspired from genomics):

- Global alignment with dynamic programming ($O(n^2)$)
- Local alignment with dynamic programming [Irving, 2004] ($O(n^2)$)
- Extending matching parts via germs fingerprinting (in $O(n)$): Greedy String Tiling

Drawback: global temporal complexity between $O(k^2 n)$ and $O(k^2 n^2)$

A better solution: global comparison

- Finding tuples of matching token sequences
- Use of global representation of the sequences extracted from the projects for easy similarity retrieval like suffix trees or suffix arrays constructed in linear time

Some implemented tools for cluster similarities detection on token sequences

- JPlag (Greedy String Tiling [Wise, 1994])
- Moss (Karp-Rabin fingerprinting with Winnowing selection [Schleimer, 2003])
- SID [Chen, 2004] (Kolgomorov's complexity approximation) : computation of the amount of shared information (similarity metrics) through a compression algorithm for each pair of projects

Common drawbacks

No open-source similarity detection tool available (security through obscurity).
Work at a file-level (file = sequence of tokens) with no consideration of the call graph.

Peculiarities and aims of our method

Peculiarities

- Global comparison of token sequences through a suffix array
- Rewriting of the call graph of the programs through shared chunks of functions (outlining through the projects)

Aims

- Improved time complexity over a pair-comparison method ($O(kn + k^2)$ for a cluster of k projects of mean size n)
- Better resistance to common methods of obfuscation (ins/del/move of instructions, {in-out}-lining, ...)

Outline

Tokenization example: a bubble-sort function

Source code

```
void subsort(int t[],int i) {
  int i_min; if (i ≥ SIZE) return;
  i_min = min_index(t,i);
  exchange(t, i, i_min); subsort(t, i+1);
}
```

Tokenization example: a bubble-sort function

Source code

```
void subsort(int t[],int i) {
  int i_min; if (i ≥ SIZE) return;
  i_min = min_index(t,i);
  exchange(t, i, i_min); subsort(t, i+1);
}
```

1. Tokenization of the source code using a lexer

Raw sequence of abstract tokens

```
IF LPAR identifier GE identifier RPAR RETURN SEMI
identifier ASSIGN <min_index> SEMI
<exchange> SEMI
<subsort> SEMI
```

Tokenization example: a bubble-sort function

Source code

```
void subsort(int t[],int i) {
  int i_min; if (i ≥ SIZE) return;
  i_min = min_index(t,i);
  exchange(t, i, i_min); subsort(t, i+1);
}
```

1. Tokenization of the source code using a lexer
2. Light syntactic analysis to extract functions and functions calls

Raw sequence of abstract tokens

```
IF LPAR identifier GE identifier RPAR RETURN SEMI
identifier ASSIGN <min_index> SEMI
<exchange> SEMI
<subsort> SEMI
```

Factorization algorithm principle

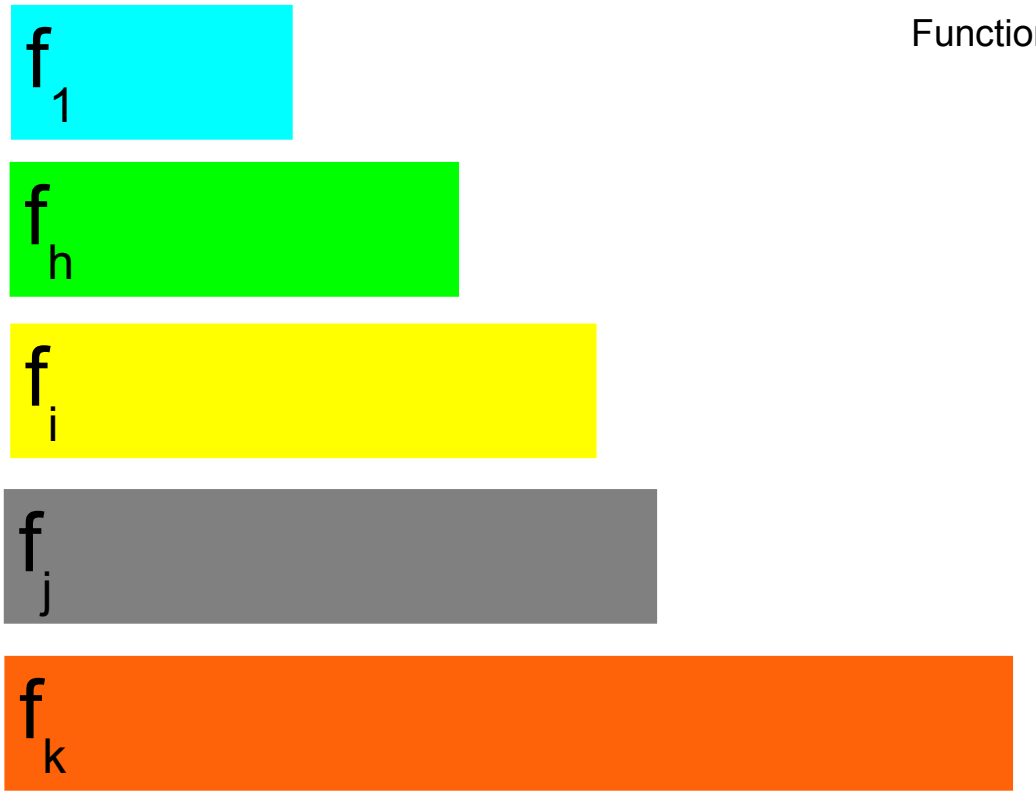
Aim of the factorization algorithm

Merging call-graphs of several projects into a global synthetic call-graph introducing shared source code as new called functions

Factorization principle

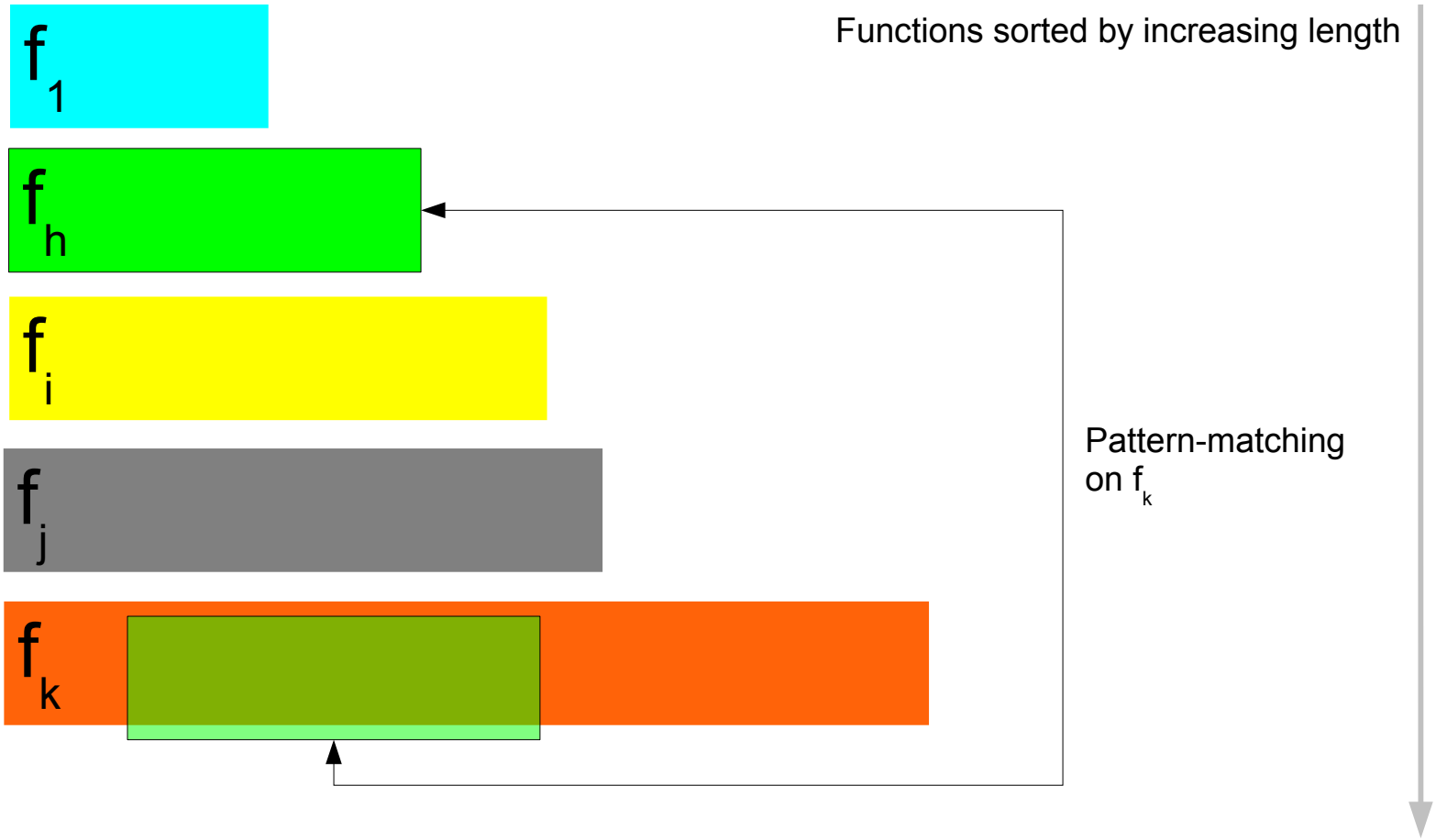
1. Sort the original functions by increasing token length
2. Factorize each function using chunks of smaller functions
3. Externalize the matched function chunks
4. Re-iterate the process for a better granularity of similarities

Factorization of f_k using $F_{<k}$

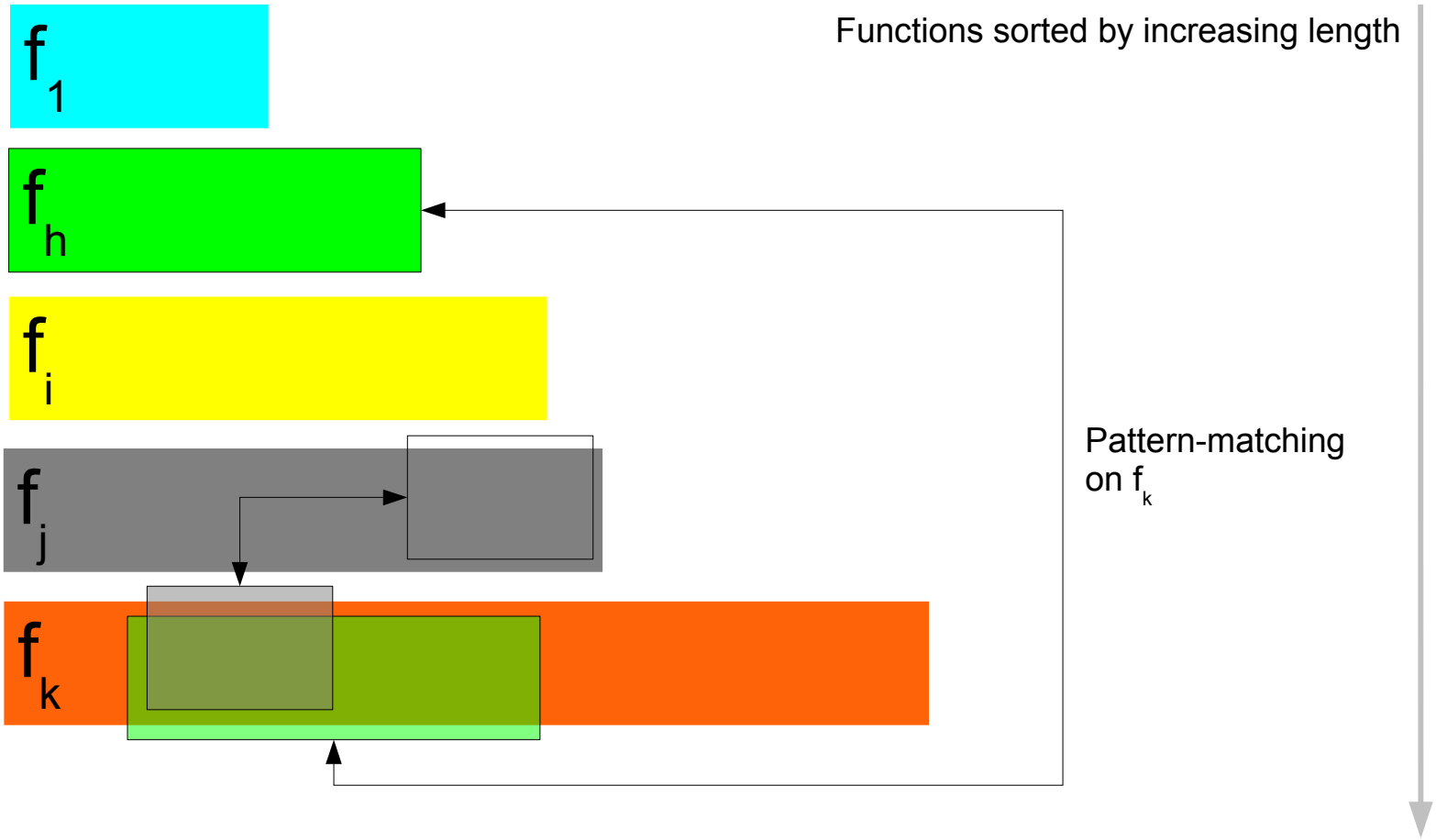


Functions sorted by increasing length

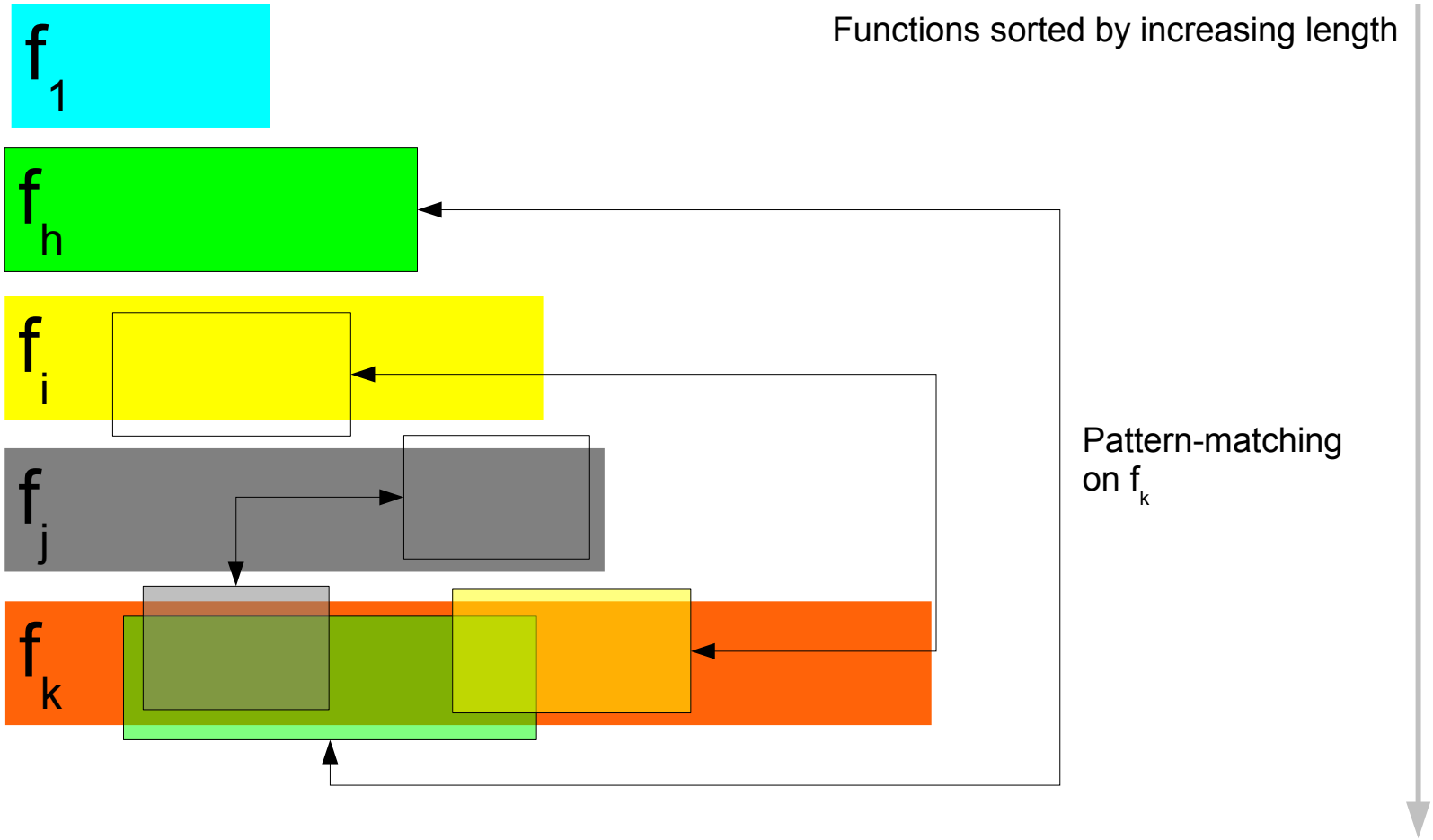
Factorization of f_k using $F_{<k}$



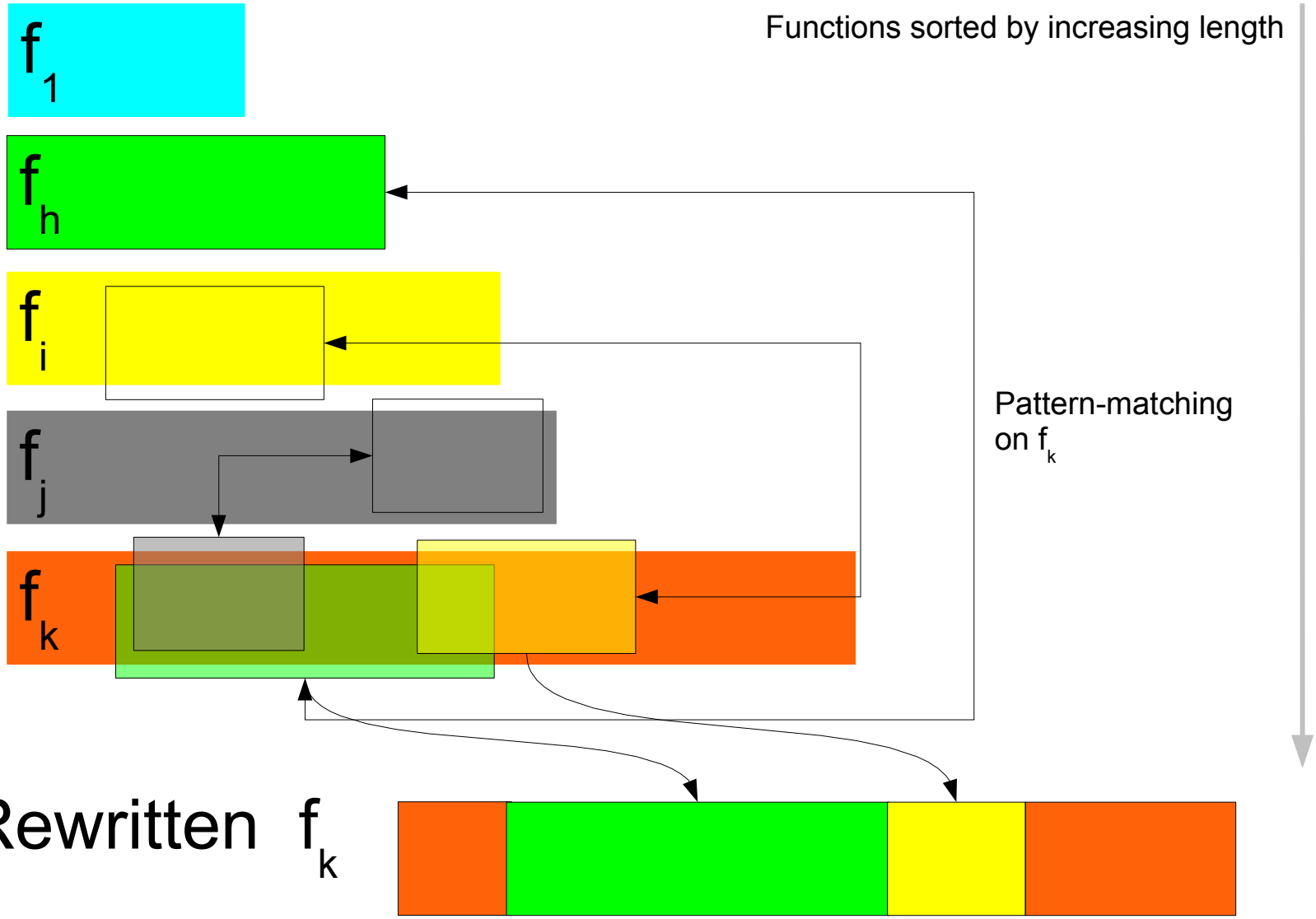
Factorization of f_k using $F_{<k}$



Factorization of f_k using $F_{<k}$



Factorization of f_k using $F_{<k}$



A word on pattern matching

Finding chunks of functions into longer function

- Use of a classical linearly built suffix array [Manber, 1990]
- New dedicated data structure to find common substrings for each position of the function: the *Enhanced Parent Interval Table*

Eliminating overlapping matches

- Selection of the most-weighted non-overlapping factors (with a priority queue and segment tree)
- Each function is the concatenation of new externalized functions

Factorization algorithm on a bubble-sort source code

Original source code with rewrite of *subsort* to *subsort_inlined*.

f_1: void exchange(int t[],int a,int b)

```
tmp = t[a]; t[a] = t[b]; t[a] = tmp;
```

f_2: void subsort(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = min_index(t,i);
exchange(t, i, i_min);
subsort(t, i+1);
```

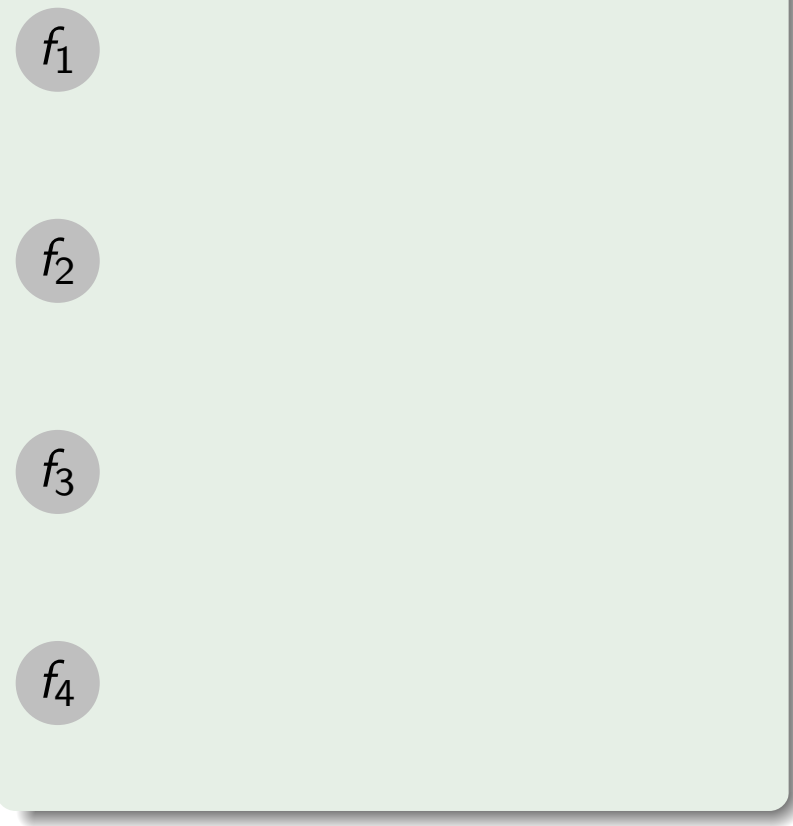
f_3: int min_index(int t[],int start)

```
i_min = start; j = start+1;
while (j<SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
return i_min;
```

f_4: void subsort_inlined(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = i; j = i+1;
while (j < SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
tmp = t[i]; t[i] = t[i_min]; t[i_min] = tmp;
subsort_inlined(t, i+1);
```

Call graph



Factorization algorithm on a bubble-sort source code

Original call graph before the run of the factorization algorithm.

f_1: void exchange(int t[],int a,int b)

```
tmp = t[a]; t[a] = t[b]; t[a] = tmp;
```

f_2: void subsort(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = min_index(t,i);
exchange(t, i, i_min);
subsort(t, i+1);
```

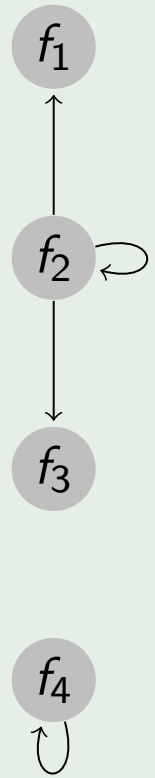
f_3: int min_index(int t[],int start)

```
i_min = start; j = start+1;
while (j<SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
return i_min;
```

f_4: void subsort_inlined(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = i; j = i+1;
while (j < SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
tmp = t[i]; t[i] = t[i_min]; t[i_min] = tmp;
subsort_inlined(t, i+1);
```

Call graph



Factorization algorithm on a bubble-sort source code

1st iteration: part of *min_index* found in *subsort_inlined* $\longrightarrow$ creation of Φ_1.

f_1: void exchange(int t[],int a,int b)

```
tmp = t[a]; t[a] = t[b]; t[a] = tmp;
```

f_2: void subsort(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = min_index(t,i);
exchange(t, i, i_min);
subsort(t, i+1);
```

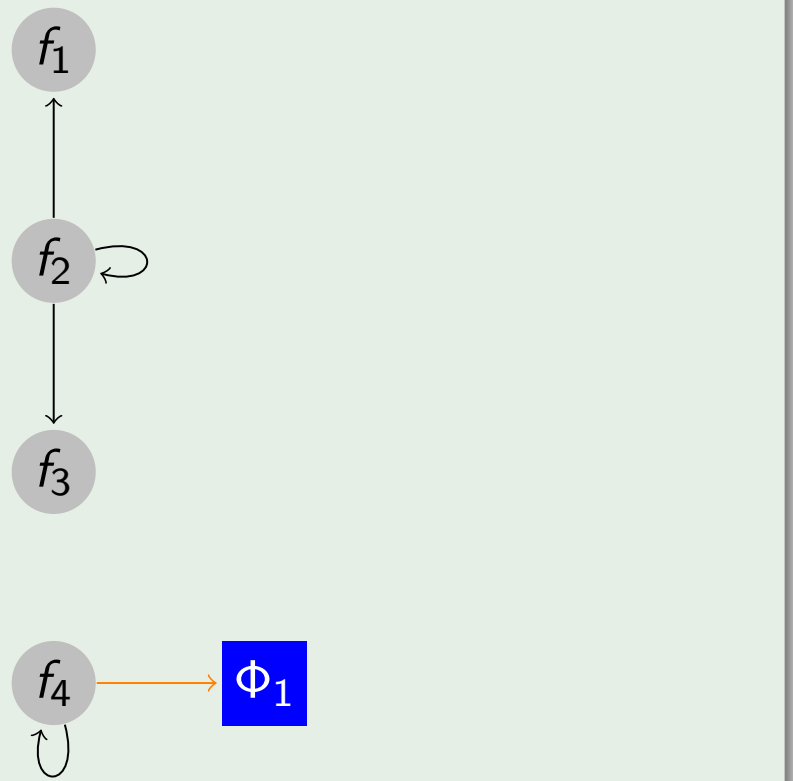
f_3: int min_index(int t[],int start)

```
i_min = start; j = start+1;
while (j<SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
return i_min;
```

f_4: void subsort_inlined(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = i; j = i+1;                                   Φ1 ←—
while (j < SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }   Φ1 ←—
tmp = t[i]; t[i] = t[i_min]; t[i_min] = tmp;
subsort_inlined(t, i+1);
```

Call graph



Factorization algorithm on a bubble-sort source code

1st iteration: *exchange* found in *subsort_inlined* $\longrightarrow$ creation of Φ_2.

f_1: void exchange(int t[],int a,int b)

```
tmp = t[a]; t[a] = t[b]; t[a] = tmp;
```

f_2: void subsort(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = min_index(t,i);
exchange(t, i, i_min);
subsort(t, i+1);
```

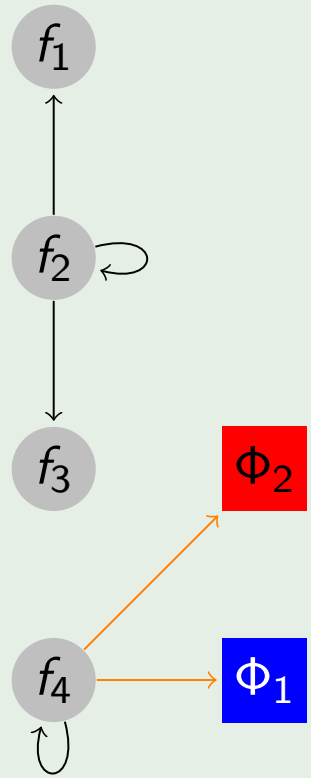
f_3: int min_index(int t[],int start)

```
i_min = start; j = start+1;
while (j<SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
return i_min;
```

f_4: void subsort_inlined(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = i; j = i+1;
while (j < SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
tmp = t[i]; t[i] = t[i_min]; t[i_min] = tmp;        Φ2 ←−
subsort_inlined(t, i+1);
```

Call graph



Factorization algorithm on a bubble-sort source code

1st iteration: externalization of remaining part of *subsort_inlined* to f_4' (no match reported with f_2 due to the token matching threshold).

f_1: void exchange(int t[],int a,int b)

```
tmp = t[a]; t[a] = t[b]; t[a] = tmp;
```

f_2: void subsort(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = min_index(t,i);
exchange(t, i, i_min);
subsort(t, i+1);
```

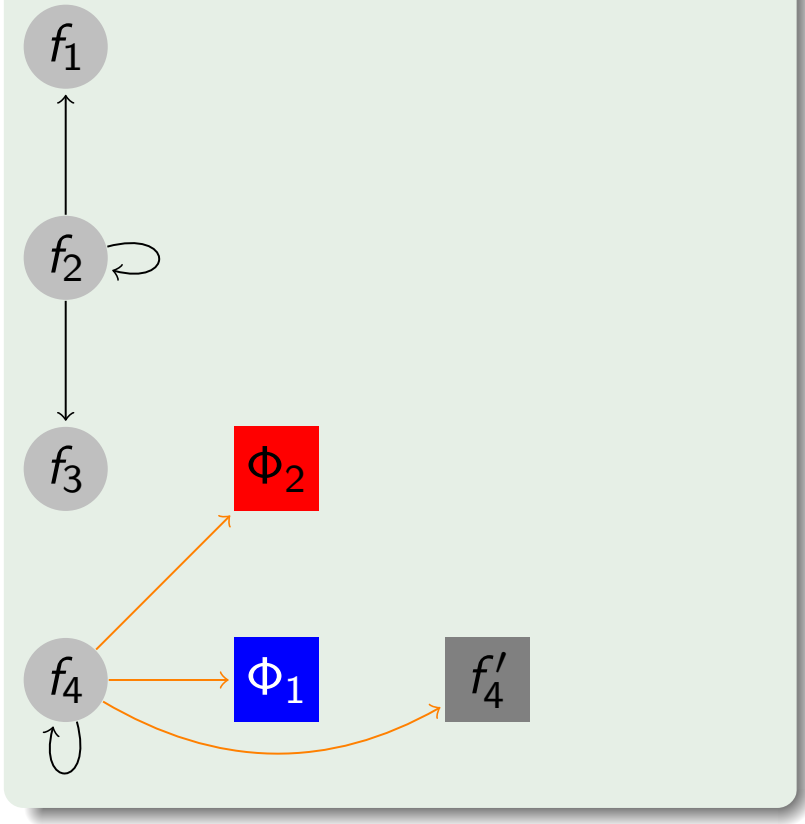
f_3: int min_index(int t[],int start)

```
i_min = start; j = start+1;
while (j<SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
return i_min;
```

f_4: void subsort_inlined(int t[],int i)

```
if (i ≥ SIZE) return;                    f4' ←
i_min = i; j = i+1;
while (j < SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
tmp = t[i]; t[i] = t[i_min]; t[i_min] = tmp;
subsort_inlined(t, i+1);
```

Call graph



Factorization algorithm on a bubble-sort source code

2nd iteration: Φ_1 matching with part of *min_index*, Φ_2 matching with *exchange*

f_1: void exchange(int t[],int a,int b)

```
tmp = t[a]; t[a] = t[b]; t[a] = tmp;      Φ2 ⟵
```

f_2: void subsort(int t[],int i)

```
if (i ≥ SIZE) return;          f2' ⟵
i_min = min_index(t,i);
exchange(t, i, i_min);
subsort(t, i+1);
```

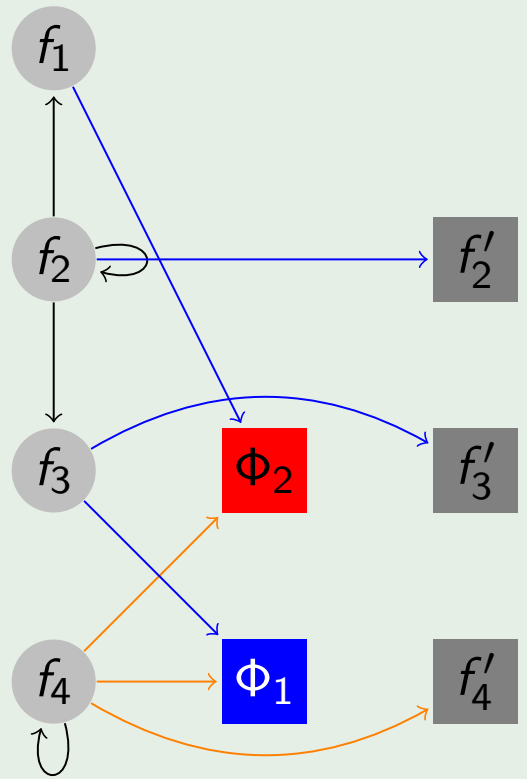
f_3: int min_index(int t[],int start)

```
i_min = start; j = start+1;                                  Φ1 ⟵
while (j<SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }       Φ1 ⟵
return i_min;                                                f3' ⟵
```

f_4: void subsort_inlined(int t[],int i)

```
if (i ≥ SIZE) return;
i_min = i; j = i+1;
while (j < SIZE) { if(t[j] < t[i_min]) i_min = j; j++; }
tmp = t[i]; t[i] = t[i_min]; t[i_min] = tmp;
subsort_inlined(t, i+1);
```

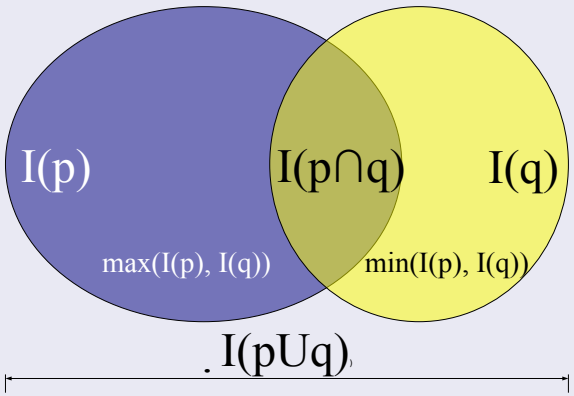
Call graph



Outline

Similarity metrics

Information amounts

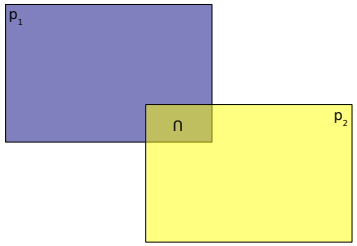


Similarity metrics

$$\begin{bmatrix} sc_{incl} \\ sc_{cover} \\ sc_{simil} \end{bmatrix}(p,q) = \frac{1}{I(p \cup q)} \begin{bmatrix} \min\left(I(p), I(q)\right) \\ \max\left(I(p), I(q)\right) \\ I(p \cap q) \end{bmatrix}$$

- Using the set of leaves reached by a project as the practical amount of information (size of these leaves)
- Key idea: similar original functions reach the same set of leaves
- No notion of order: metrics insensitive to code moves
- Low pattern-matching threshold $\longrightarrow$ low precision

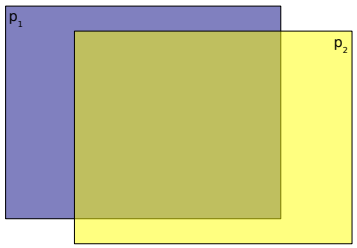
Similarity cases



Small chunks of code shared between p_1 and p_2

$|If(p_1)| \simeq |If(p_2)|$

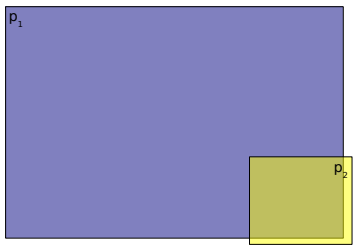
$sc_{simil}(p_1, p_2) << sc_{cover}(p_1, p_2) \simeq sc_{incl}(p_1, p_2)$



p_1 and p_2 are two quasi-clones

$If(p_1) \simeq If(p_2)$

$sc_{simil}(p_1, p_2) \simeq sc_{cover}(p_1, p_2) \simeq sc_{incl}(p_1, p_2) \simeq 1$



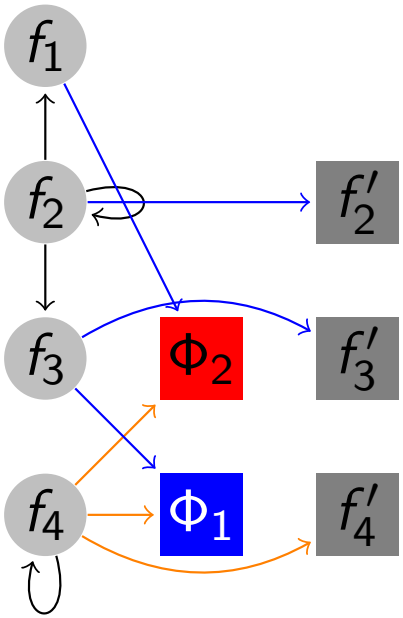
Small project p_2 is a plagiarism of a larger project p_1

$If(p_2) \subset If(p_1)$

$sc_{simil}(p_1, p_2) \simeq sc_{cover}(p_1, p_2) << sc_{incl}(p_1, p_2) \simeq 1$

Similarity metrics applied on the bubble-sort example

- Reachable leaves matrix (computed from the call-graph):



	Φ_1	Φ_2	f'_2	f'_3	f'_4
f_1(exchange)	0	1	0	0	0
f_2(subsort)	1	1	1	1	0
f_3(min_index)	1	0	0	1	0
f_4(subsort_inlined)	1	1	0	0	1

- Similarity matrix ([sc_{simil}, sc_{cover}, sc_{incl}]):

	f_1	f_2	f_3	f_4
f_1				
f_2	[0.35, 0.35, 1.00]			
f_3	[0.00, 0.00, 0.00]	[0.51, 0.51, 1.00]		
f_4	[0.36, 0.36, 1.00]	[0.77, 0.86, 0.88]	[0.52, 0.52, 1.00]	

Outline

Test conditions

Test against *Moss* and *JPlag* with a set of 36 student projects ($\sim$ 600 lines of ANSI C).
Match report threshold=10 abstract tokens

Obfuscation methods

- Identifier substitutions (insensitivity with token abstraction)
- Move of functions
- Insertion and deletion of useless small instructions
- Insertion of repetitive useless blocks

Similarity results

Kind of obfuscation	*Clone*	*Id. subst.*	*Moves*	*Ins/del*	*Inlining*	*Outlining*	*Flooding*	*Non plagiarized*
Factorization (sc_{incl})	1.0	1.0	1.0	0.72	0.87	0.81	0.84	0.04
Moss	0.94	0.94	0.90	0.25	0.74	0.56	0.73	0.01
JPlag	0.99	0.99	0.97	0.37	0.87	0.86	0.81	0.00

Figure: Plagiarism scores (sc_{incl}) between the original and the obfuscated projects

Linear experimental temporal complexity

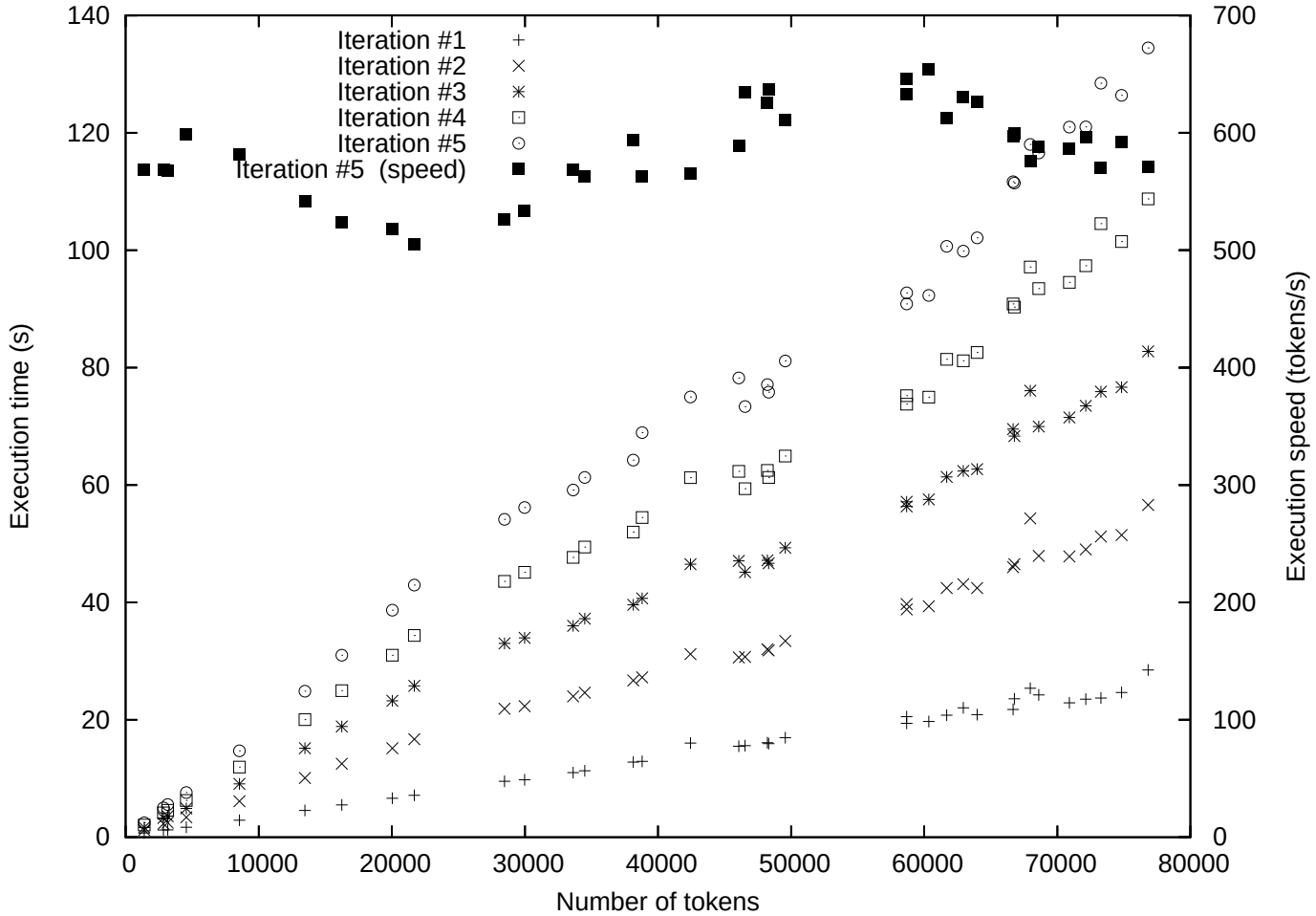


Figure: Running times of our factorization method applied on the test-set (test environment: Sun JRE 1.5 Client VM, P4 3 Ghz, 1 Gb RAM)

Outline

To conclude: future works

- Implementing an incremental factorization algorithm (extension to database indexation)
- Introducing flow analysis to increase precision (requires syntactic analysis)
- Developing a human-friendly visualization method for the results (smart interactive graph-call visualization)

Thank you for your attention

Questions?

References I


Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna and L. Bier
Clone detection using abstract syntax trees
ICSM (1998).


Krinke, J.
Identifying similar code with program dependence graphs
WCRE, 2001.


Irving, R.
Plagiarism and collusion detection using the Smith-Waterman algorithm (2004).


Wise, M. J.
Running karp-rabin matching and greedy string tiling
Technical Report 463, Dep. of Comp. Sci., Sidney Univ. (1994).

References II

Schleimer, S., D. S. Wilkerson and A. Aiken
Winnowing: local algorithms for document fingerprinting
A. P. NY, editor, *Proc. Int. Conf. on Management of Data*, 2003.

Chen, X., B. Francia, M. Li, B. McKinnon and A. Seker
Shared information and program plagiarism detection
IEEE Trans. Information Theory (2004).

Manber, U. and G. Myers
Suffix arrays: a new method for on-line string searches
Soc. for Industrial and Applied Math. Philadelphia, PA, USA, 1990.