Mining Edge-Weighted Call Graphs to Localise Software Bugs

Frank Eichinger Klemens Böhm Matthias Huber

Institute for Program Structures and Data Organisation (IPD)
Universität Karlsruhe (TH), Germany

16th September 2008

Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Debugging is time consuming and expensive.
  - Automated localisation is needed!

Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Debugging is time consuming and expensive.
  - Automated localisation is needed!

- Idea:
  - Locate bugs with **data mining** techniques.

Locating Bugs in Software

- Software is almost never shipped bug-free, even if tested extensively.
- Debugging is time consuming and expensive.
  - Automated localisation is needed!

- Idea:
  - Locate bugs with **data mining** techniques.
  - Using a **weighted graph mining** approach.

Outline

Data Mining in Software Engineering

- Traditional data mining techniques process feature vectors of numerical and categorical data.
- In software engineering, e.g., code metrics have been used to build classifiers.

Data Mining in Software Engineering

- Traditional data mining techniques process feature vectors of numerical and categorical data.
- In software engineering, e.g., code metrics have been used to build classifiers.

- Emerging idea:
  - Look at **program executions** represented as **call graphs** and analyse the graph structure.
  - Identify substructures typical for failing executions.

Data Mining in Software Engineering

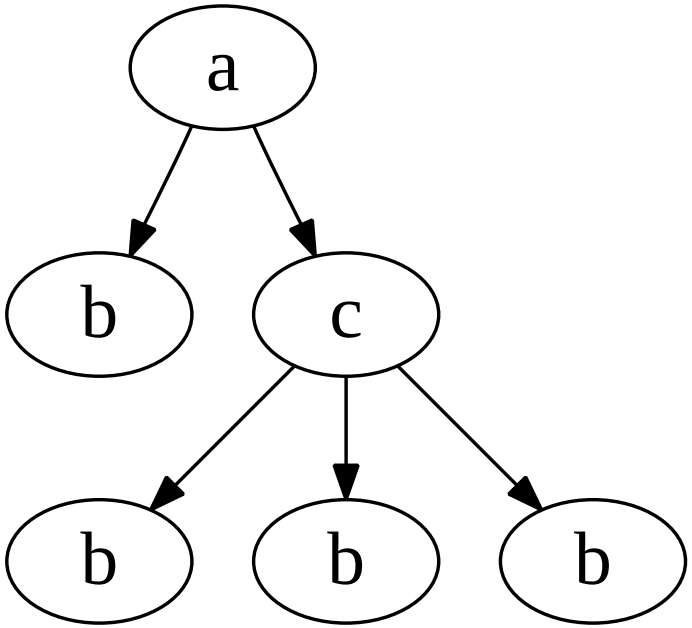
- Traditional data mining techniques process feature vectors of numerical and categorical data.
- In software engineering, e.g., code metrics have been used to build classifiers.

- Emerging idea:
  - Look at **program executions** represented as **call graphs** and analyse the graph structure.
  - Identify substructures typical for failing executions.

- New aspect:
  - Explicitly analyse **call frequencies** (**edge weights**) beside graph structures.

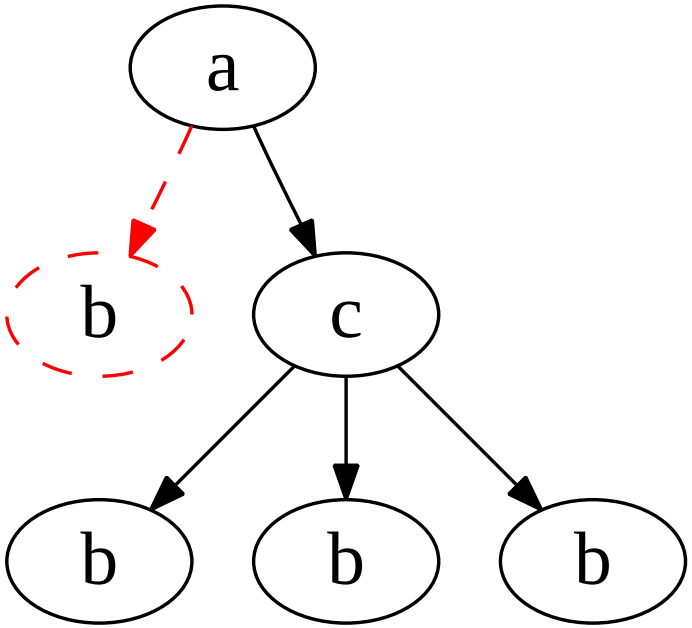
Call Graphs

- Program executions as call graphs:
  - Methods → nodes
  - Method calls → edges



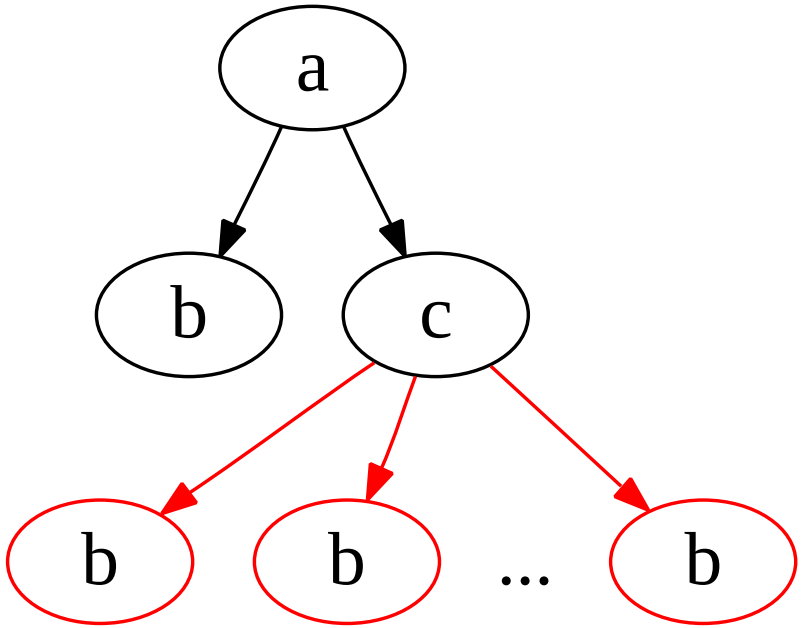
Call Graphs

- Program executions as call graphs:
  - Methods → nodes
  - Method calls → edges
- Bugs in the call graph:
  - **Structure affecting** (existing approaches)
    - E.g., a bug in an `if`-condition in **a**

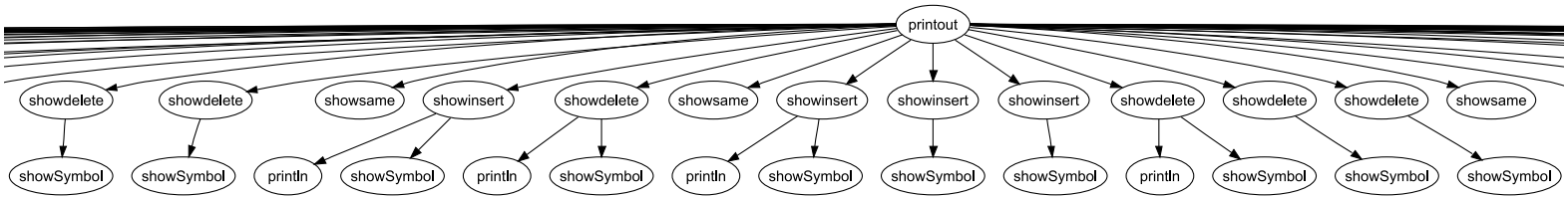


Call Graphs

- Program executions as call graphs:
  - Methods $\rightarrow$ nodes
  - Method calls $\rightarrow$ edges
- Bugs in the call graph:
  - **Structure affecting** (existing approaches)
    - E.g., a bug in an `if`-condition in **a**
  - **Call frequency affecting** (new in our contribution)
    - E.g., a bug in a loop-condition in **c**

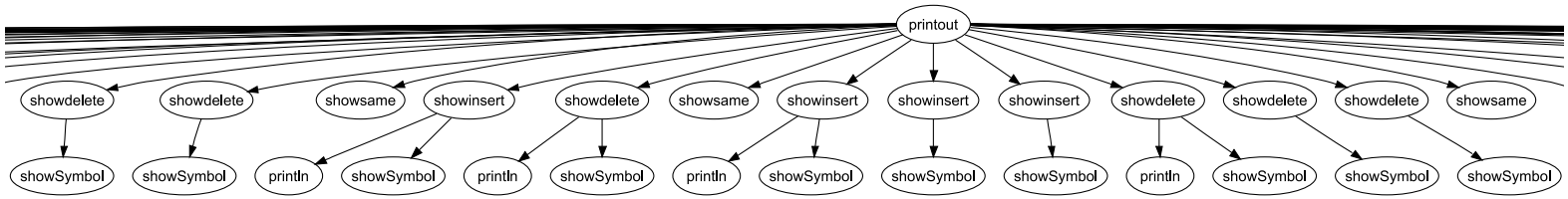


Reduction of Call Graphs

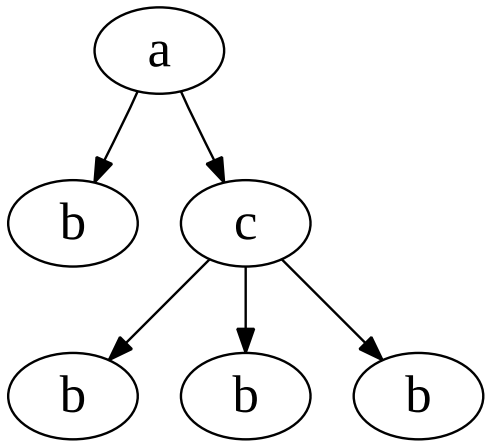


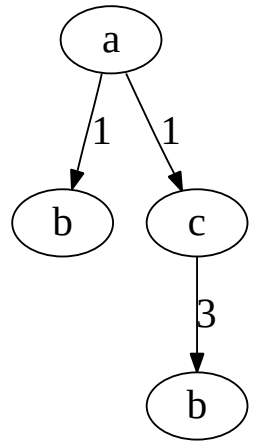
- Millions of method calls are very common!

Reduction of Call Graphs



- Millions of method calls are very common!
- Reduction of call graphs is necessary.





- Weights have not been used in previous reductions.

Weighted Graph Mining

- No algorithm available for **weighted graph mining** (as well as no problem formulation).

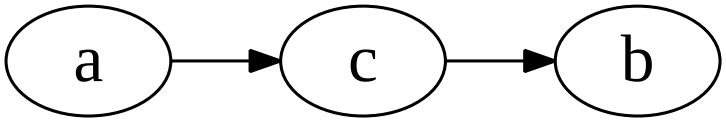
Weighted Graph Mining

- No algorithm available for **weighted graph mining** (as well as no problem formulation).
- How to make use of edge weights?
  - Preprocessing
    - Discretisation of weights
  - Postprocessing (our approach)
    - Graph mining without weights
    - Subsequent detailed analysis of weights

Finding Well Discriminating Edge Weights

1. Apply **frequent subgraph mining** to **reduced call graphs** of *correct* and *failing* program executions (ignore the weights in this step)
2. Consider only subgraphs occurring in both, *correct* and *failing* executions
3. Analyse the edge weights

- Example graph found by frequent subgraph mining:

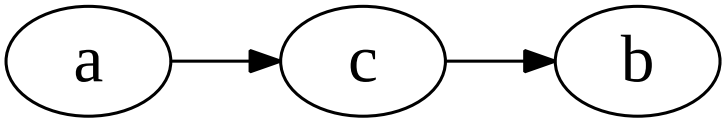


  - Average weight of $a \rightarrow c$ in *correct* executions: 1
  - Average weight of $a \rightarrow c$ in *failing* executions: 1
  - Average weight of $c \rightarrow b$ in *correct* executions: 3
  - Average weight of $c \rightarrow b$ in *failing* executions: 30

Finding Well Discriminating Edge Weights

1. Apply **frequent subgraph mining** to **reduced call graphs** of *correct* and *failing* program executions (ignore the weights in this step)
2. Consider only subgraphs occurring in both, *correct* and *failing* executions
3. Analyse the edge weights

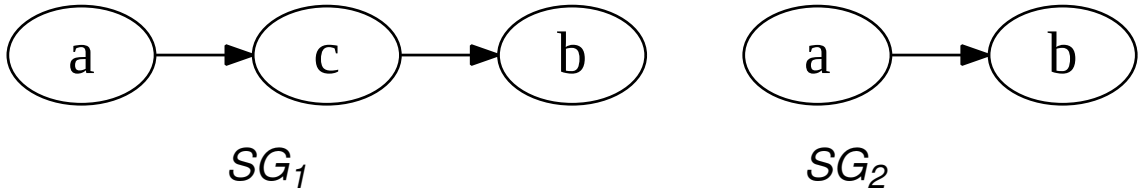
- Example graph found by frequent subgraph mining:



  - Average weight of $a \rightarrow c$ in *correct* executions: 1
  - Average weight of $a \rightarrow c$ in *failing* executions: 1
  - **Average weight of $c \rightarrow b$ in *correct* executions:** 3
  - **Average weight of $c \rightarrow b$ in *failing* executions:** 30

Entropy Based Scoring

- Assemble a table which contains every edge in every frequent subgraph:

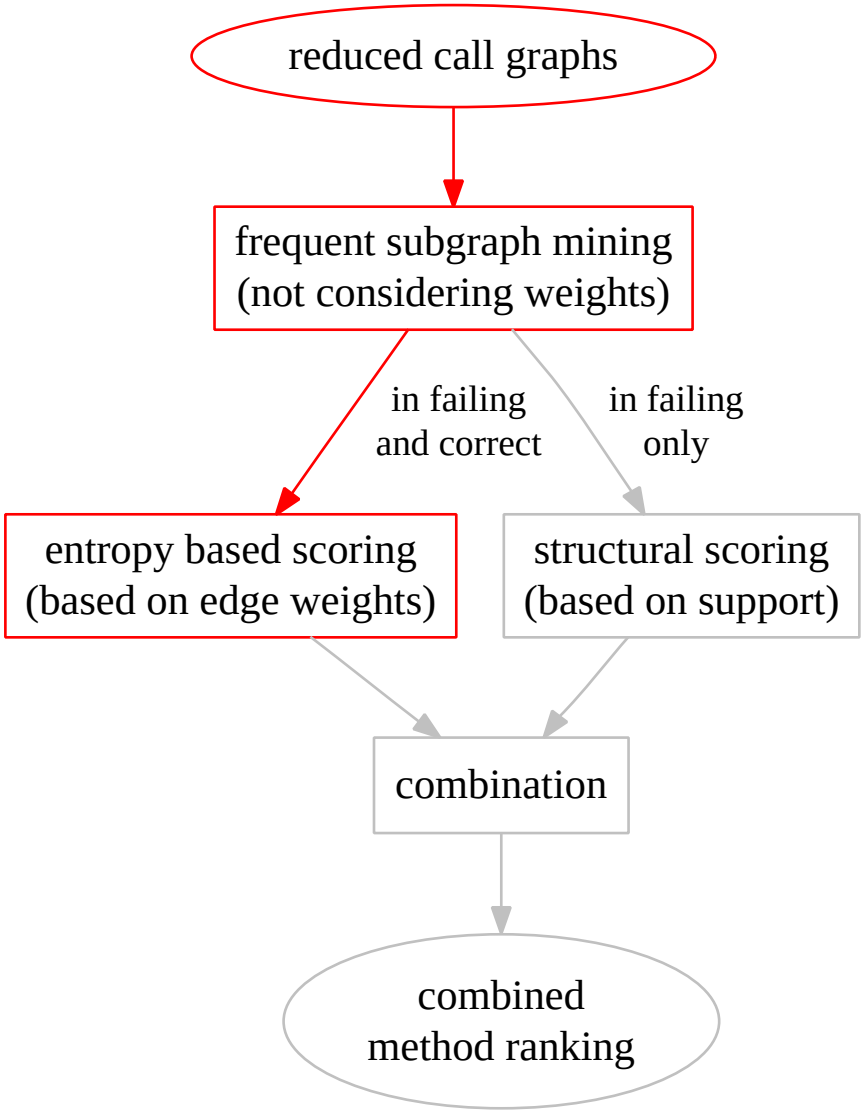


SG_1 SG_2

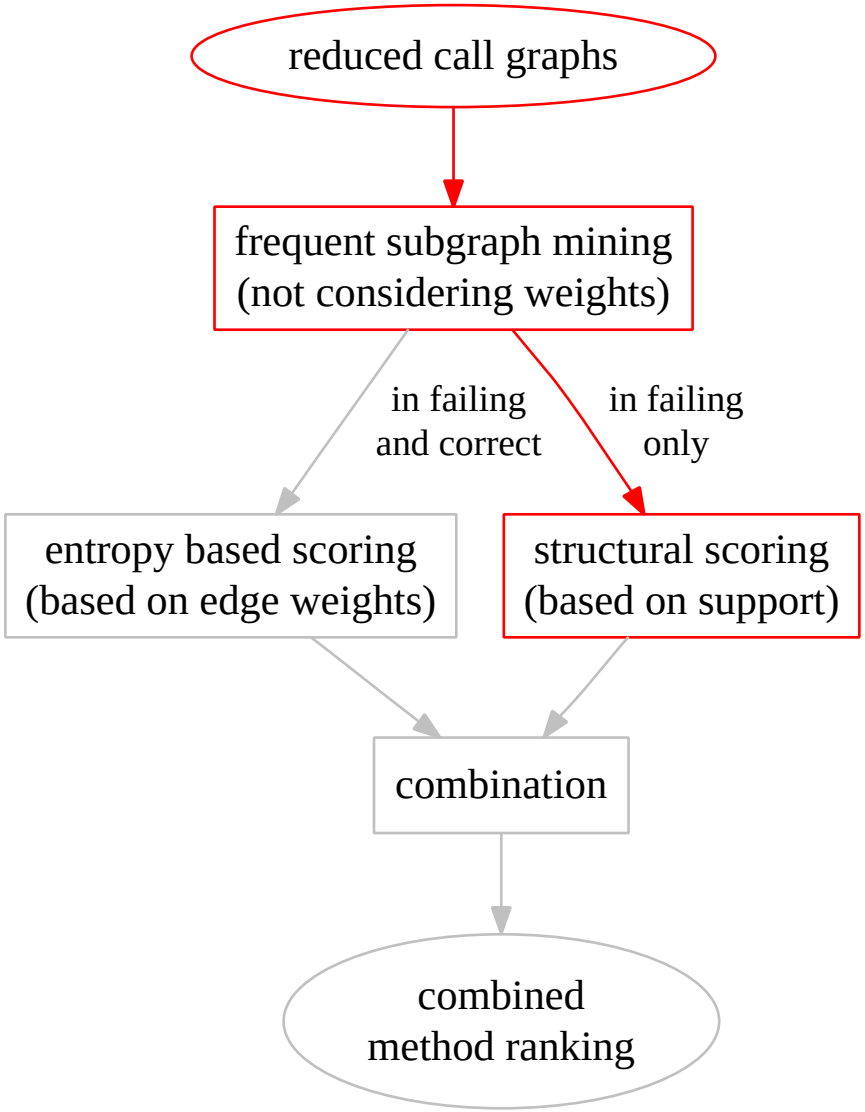
	SG_1 $a \rightarrow c$	SG_1 $c \rightarrow b$	SG_2 $a \rightarrow b$	$\cdots$	*Class*
$Execution_1$	1	30	6	$\cdots$	*failing*
$Execution_2$	1	3	7	$\cdots$	*correct*
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

- Application of the Information Gain feature selection algorithm, based on entropy.
- Result: Ranking of the columns (edges)

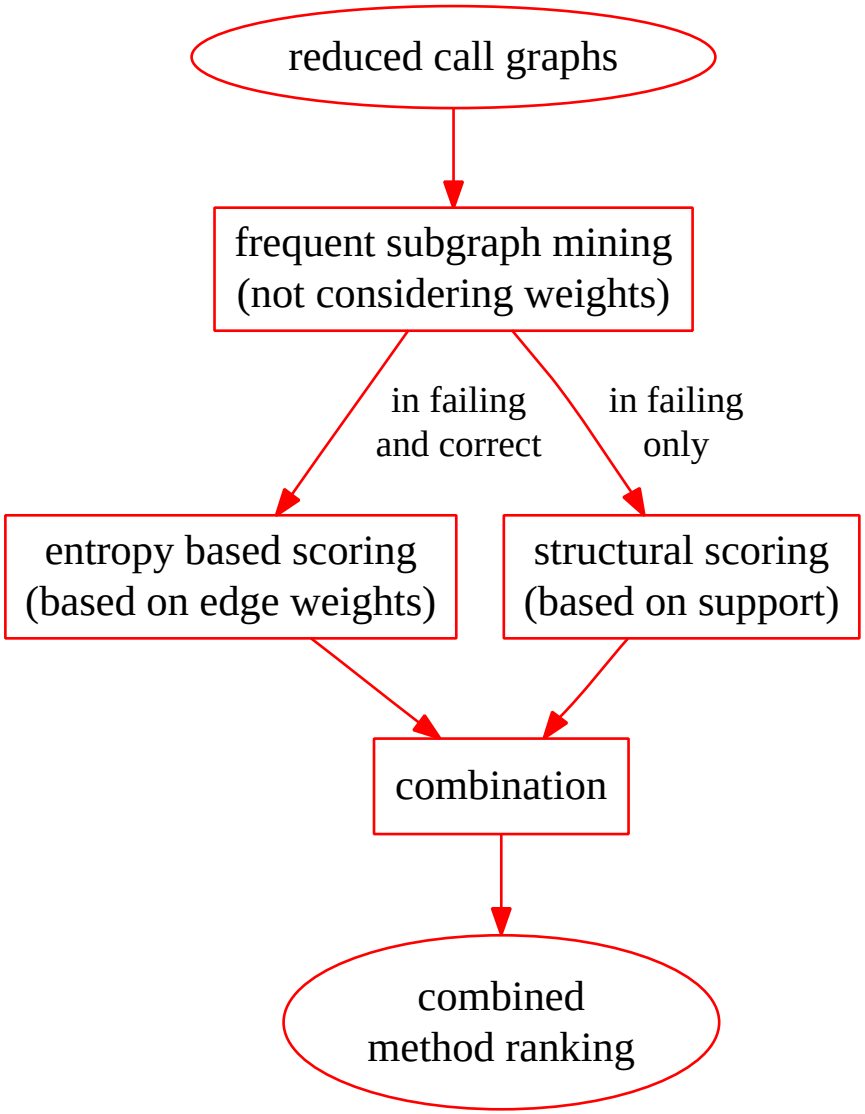
Integration of Structural Evidence



Integration of Structural Evidence



Integration of Structural Evidence



- Usage of two kinds of evidence: **frequency** (left) and **structure** (right)
- Both types of bugs can be located: **call frequency** and **structure affecting** ones

Results

- Example output:

	METHOD	SCORE
1	`inputscan()`	0.9833
2	`showinsert()`	0.9204
3	`showdelete()`	0.4876
4	`oldconsume()`	0.4876
5	`addSymbol()`	0.2428

Results

- Example output:

	METHOD	SCORE
1	`inputscan()`	0.9833
2	**`showinsert()`**	**0.9204**
3	`showdelete()`	0.4876
4	`oldconsume()`	0.4876
5	`addSymbol()`	0.2428

- The bug was instrumented in `showinsert()`.
- A software developer has to check two methods only.

Experiments

- Setting:
  - Open source Java tool
  - 14 instrumented bugs
    (same types of bugs as in related publications)
- Results (compared to related work):
  - Novel ability to detect call frequency affecting bugs
  - Doubled precision of bug localisations

Conclusion

- Summary of contributions
  - New call graph reduction technique
  - Knowledge discovery in weighted classified graphs
    - Numerical technique subsequent to structural approach
  - **Finding:**
    The combined analysis of numerical and structural evidence is key for precise results.
- Current and future work
  - Weight based constraints
  - Mining of large graphs/large software projects
  - Other fields of application

Questions?

Thank you for your attention!

CU @ my poster.