Identifying Bug Signatures Using Discriminative Graph Mining

Hong Cheng[1], David Lo[2],
Yang Zhou[1], Xiaoyin Wang[3],
and Xifeng Yan[4]

[1]Chinese University of Hong Kong
[2]Singapore Management University
[3]Peking University
[4]University of California at Santa Barbara

Automated Debugging

- Bugs part of day-to-day software development
- Bugs caused the loss of much resources
  - NIST report 2002
  - 59.5 billion dollars/annum
- Much time is spent on debugging
  - Need support for debugging activities
  - Automate debugging process
- Problem description
  - Given labeled correct and faulty execution traces
  - Make debugging an easier task to do

Bug Localization and Signature Identification

- Bug localization
  - Pinpointing a single statement or location which is likely to contain bugs
  - Does not produce the bug context
- Bug signature mining [Hsu et al., ASE'08]
  - Provides the context where a bug occurs
  - Does not assume "perfect bug understanding"
  - In the form of sequences of program elements
  - Occur when the bug is manifested

Outline

- Motivation: Bug Localization and Bug Signature
- Pioneer Work on Bug Signature Mining
- Identifying Bug Signatures Using Discriminative Graph Mining
- Experimental Study
- Related Work
- Conclusions and Future Work

Pioneer Work on Bug Signature Identification

- RAPID [Hsu et al., ASE'08]
  - Identify relevant **suspicious program elements** via Tarantula

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}$$

  - Compute the **longest common subsequences** that appear in all faulty executions with a sequence mining tool **BIDE** [Wang and Han, ICDE'04]
  - Sort returned signatures by length
  - Able to identify a bug involving path-dependent fault

Software Behavior Graphs

- Model software executions as behavior graphs
  - Node: method or basic block
  - Edge: call or transition (basic block/method) or return
  - Two levels of granularities: method and basic block
- Represent signatures as discriminating subgraphs

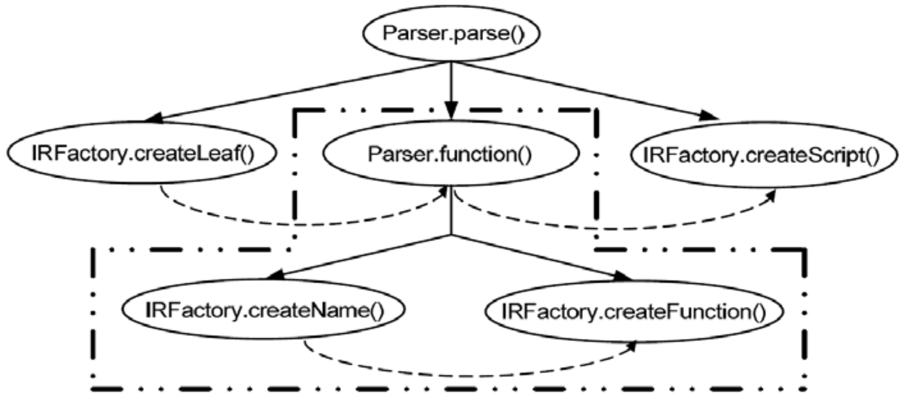
- Advantages of graph over sequence representation
  - Compactness: loops → mining scalability
  - Expressiveness: partial order and total order

Example: Software Behavior Graphs

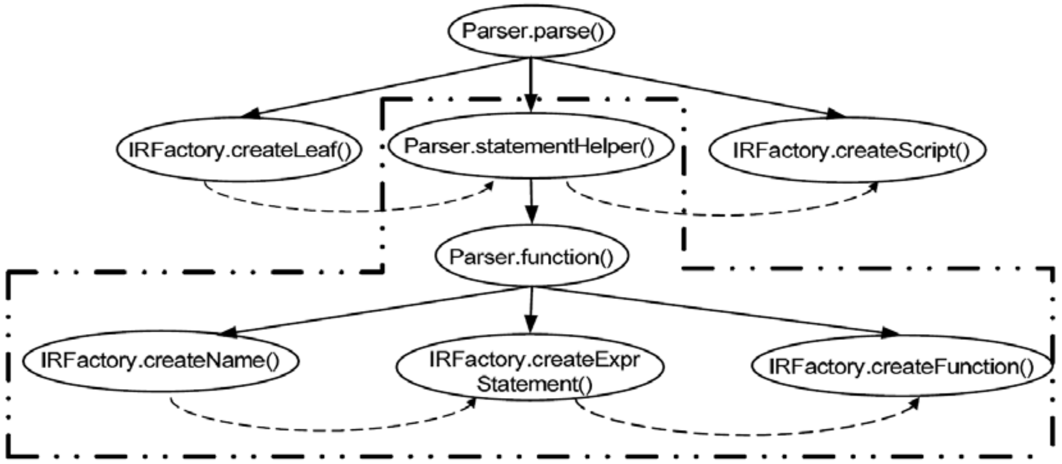
Two executions from Mozilla Rhino with a bug of number 194364

Solid edge: function call

Dashed edge: function transition



(a) Partial Behavior Graph for a Correct Execution



(b) Partial Behavior Graph for an Erroneous Execution

Bug Signature: Discriminative Sub-Graph

- Given two sets of graphs: correct and failing
- Find the most discriminative subgraph
- Information gain: $IG(c|g) = H(c) - H(c|g)$
  - Commonly used in data mining/machine learning
  - Capacity in distinguishing instances from different classes
  - Correct vs. Failing
- Meaning:
  - As frequency difference of a subgraph g in faulty and correct executions increases
  - The higher is the information gain of g
- Let F be the objective function (i.e., information gain), compute:

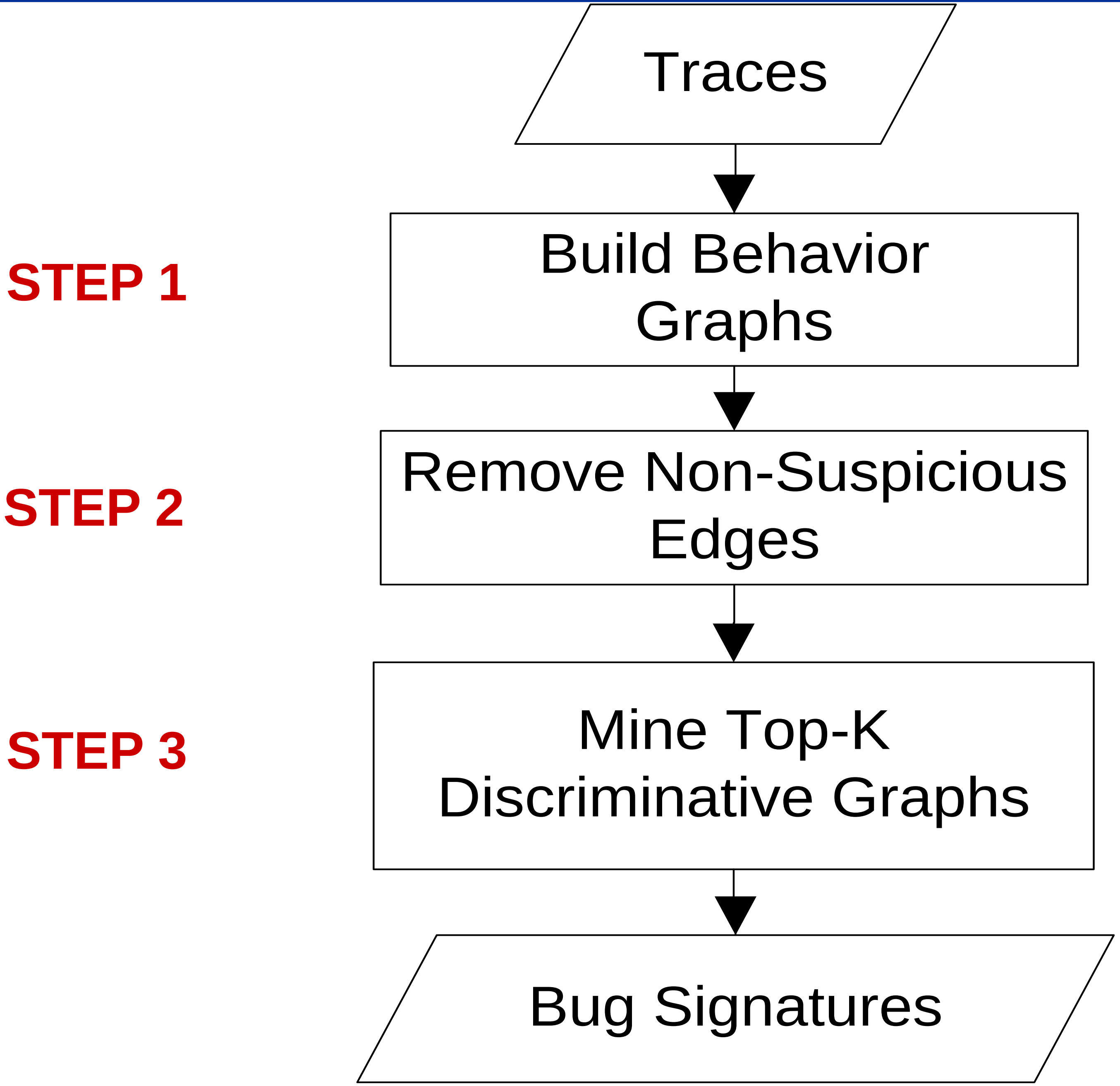
$$\arg\max_g F(g)$$

Bug Signature: Discriminative Sub-Graph

- The discriminative subgraph mined from behavior graphs contrasts the program flow of correct and failing executions and provides context for understanding the bug

- Differences with RAPID:
  - Not only element-level suspiciousness, signature-level suspiciousness/discriminative-ness
  - Does not restrict that the signature must hold across all failing executions
  - Sort by level of suspiciousness

System Framework



System Framework (2)

- Step 1
  - Trace is "coiled" to form behavior graphs
  - Based on transitions, call, and return relationship
  - Granularity: method calls, basic blocks
- Step 2
  - Filter off non-suspicious edges
  - Similar to Tarantula suspiciousness
  - Focus on **relationship** between blocks/calls

$$susp_{edg} = \frac{failed(edg)}{passed(edg)} > \frac{totalfailed}{totalpassed}$$

- Step 3
  - Mine top-k discriminating graphs
  - Distinguishes buggy from correct executions

An Example

```
1: void replaceFirstOccurrence (char arr [], int len, char cx,
                                                   char cy, char cz) {
           int i;
2:         for (i=0;i<len;i++) {
3:             if (arr[i]==cx){
4:                 arr[i] = cz;
5:                 // a bug, should be a break;
6:             }
7:             if (arr[i]==cy)){
8:                 arr[i] = cz;
9:                // a bug, should be a break;
10:            }
11:        }}
```

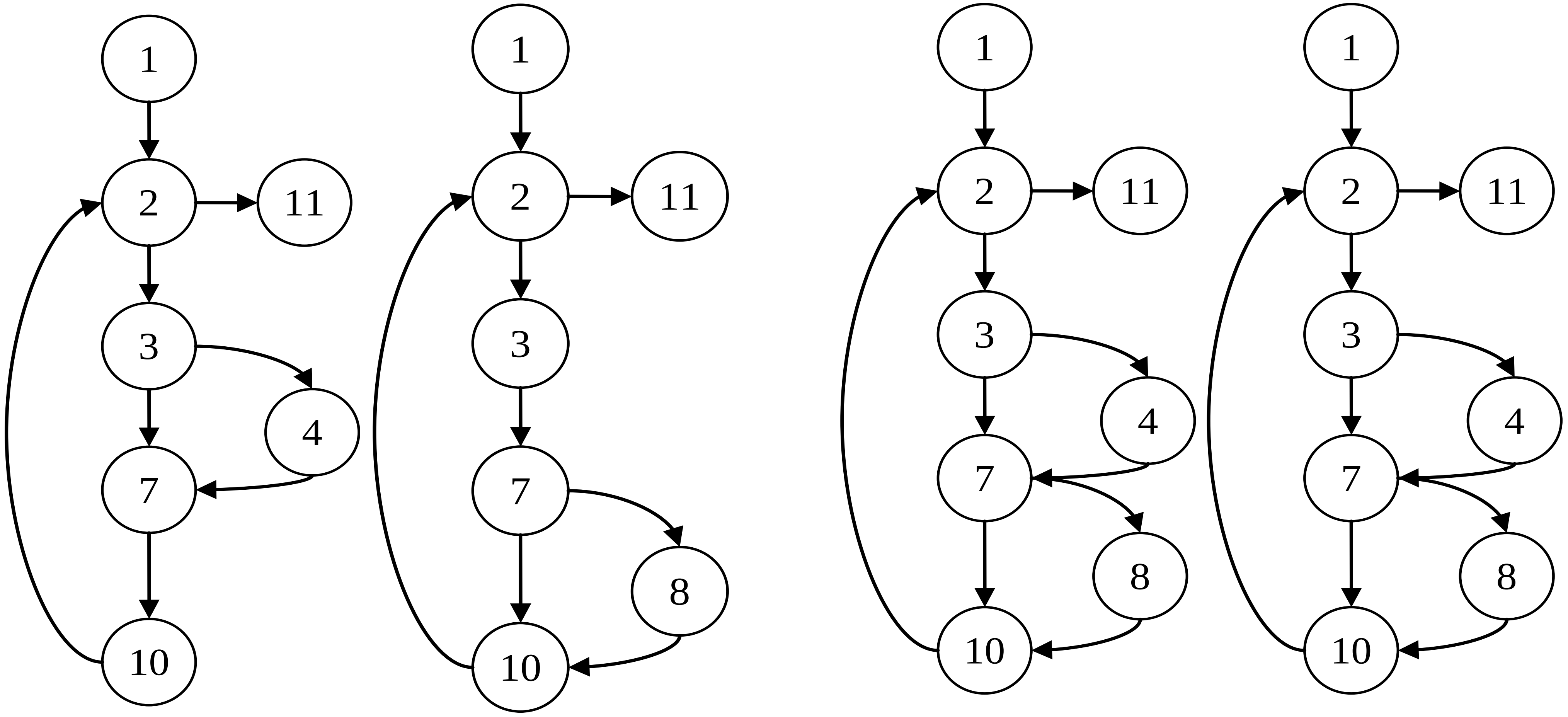
No	arr	sx	sy	sz
1	{a, b}	a	g	1
2	{a, b}	g	a	1
3	{a, g}	a	g	1
4	{a, g}	g	a	1

Four test cases

No	Trace
1	h1, 2, 3, 4, 7, 10, 2, 3, 7, 10, 11i
2	h1, 2, 3, 7, 10, 2, 3, 7, 8, 10, 11i
3	h1, 2, 3, $\boxed{4}$, 7, 10, 2, 3, 7, $\boxed{8}$, 10, 11i
4	h1, 2, 3, 7, $\boxed{8}$, 10, 2, 3, $\boxed{4}$, 7, 10, 11i

Generated traces

An Example (2)



Normal

Buggy

Behavior Graphs for Trace 1, 2, 3 & 4

An Example (3)

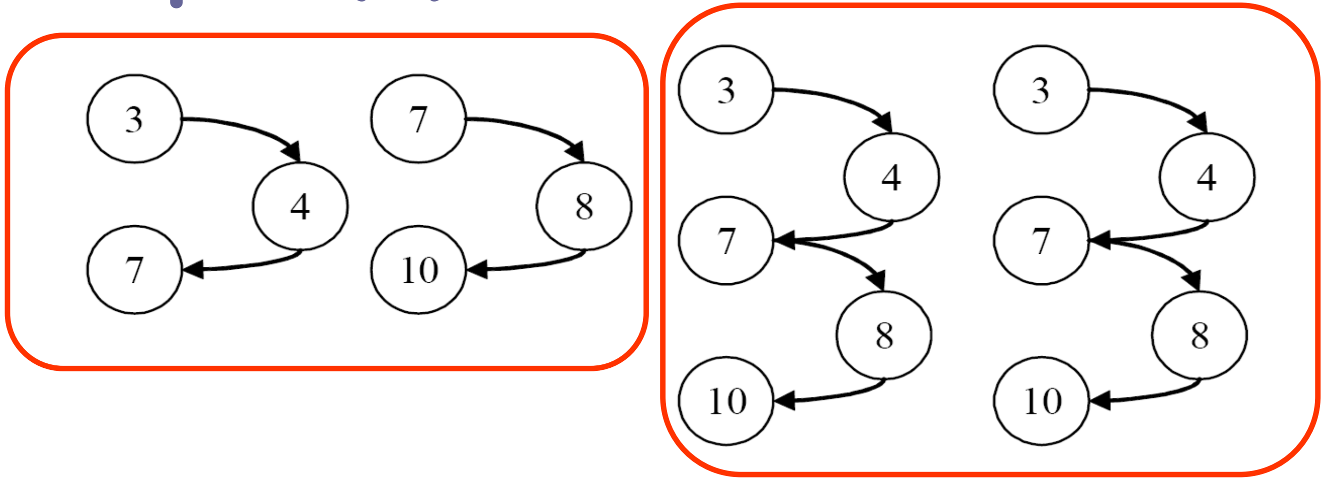


Figure 4: Pre-processed graphs for the four execution traces. All edges are labeled as trans.

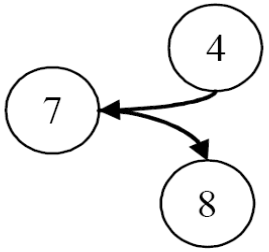
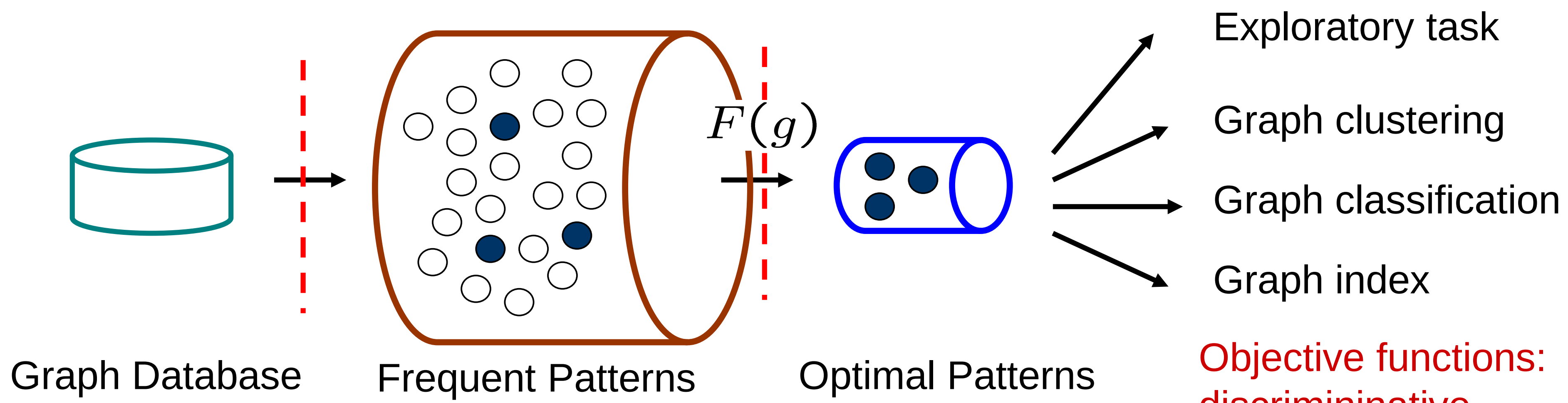


Figure 5: The discriminative subgraph. All edges are labeled as trans.

Challenges in Graph Mining: Search Space Explosion

- If a graph is frequent, all its subgraphs are frequent
  - the Apriori property
- An n-edge frequent graph may have up to 2n subgraphs which are also frequent
- Among 423 chemical compounds which are confirmed to be active in an AIDS antiviral screen dataset, there are around 1,000,000 frequent subgraphs if the minimum support is 5%

Traditional Frequent Graph Mining Framework



Objective functions: discrimininative, selective clustering tendency

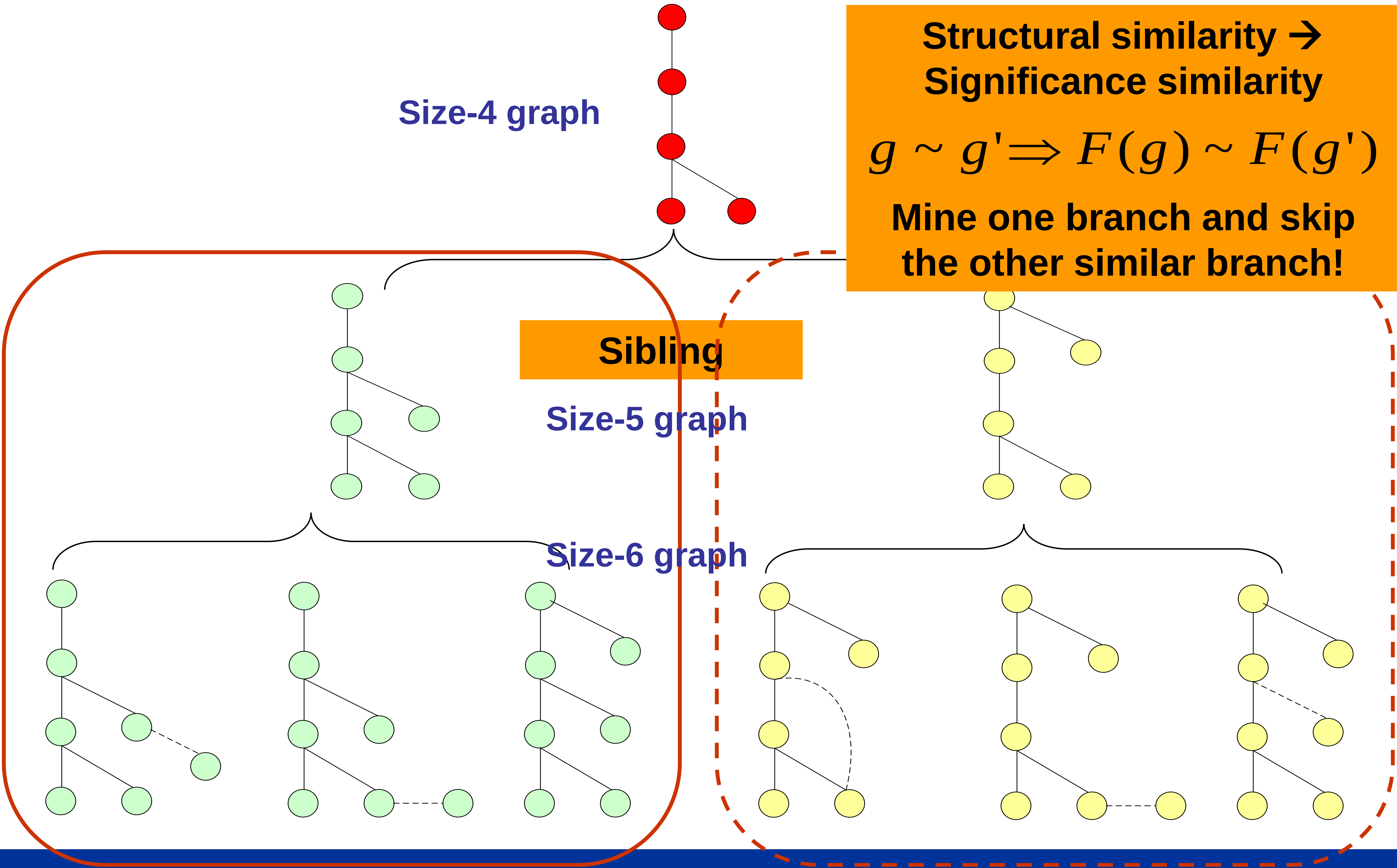
1. Computational bottleneck : millions, even billions of patterns

2. No guarantee of quality

Leap Search for Discriminative Graph Mining

- Yan et al. proposed a new leap search mining paradigm in SIGMOD'08
  - Core idea: structural proximity for search space pruning
- Directly outputs the most discriminative subgraph, highly efficient!

Core Idea: Structural Similarity



Structural similarity → Significance similarity

$$g \sim g' \Rightarrow F(g) \sim F(g')$$

Mine one branch and skip the other similar branch!

Structural Leap Search Criterion

Skip g' subtree if

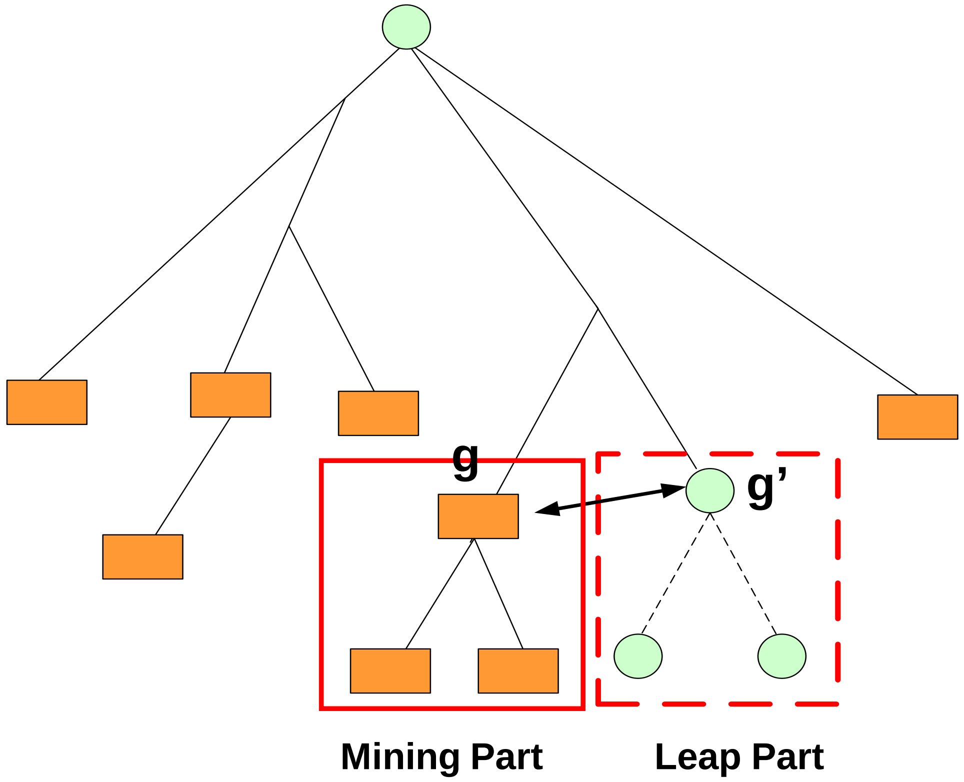
$$\frac{2\Delta_+(g,g')}{\sup_+(g)+\sup_+(g')} \le \sigma$$

$$\frac{2\Delta_-(g,g')}{\sup_-(g)+\sup_-(g')} \le \sigma$$

σ **:** tolerance of frequency dissimilarity

g : a discovered graph

g': a sibling of g



Extending LEAP to Top-K LEAP

- LEAP returns the single most discriminative subgraph from the dataset

- A ranked list of k most discriminative subgraphs is more informative than the single best one

- Top-K LEAP idea
  - The LEAP procedure is called for k times
  - Checking partial result in the process
  - Producing k most discriminative subgraphs

Experimental Evaluation

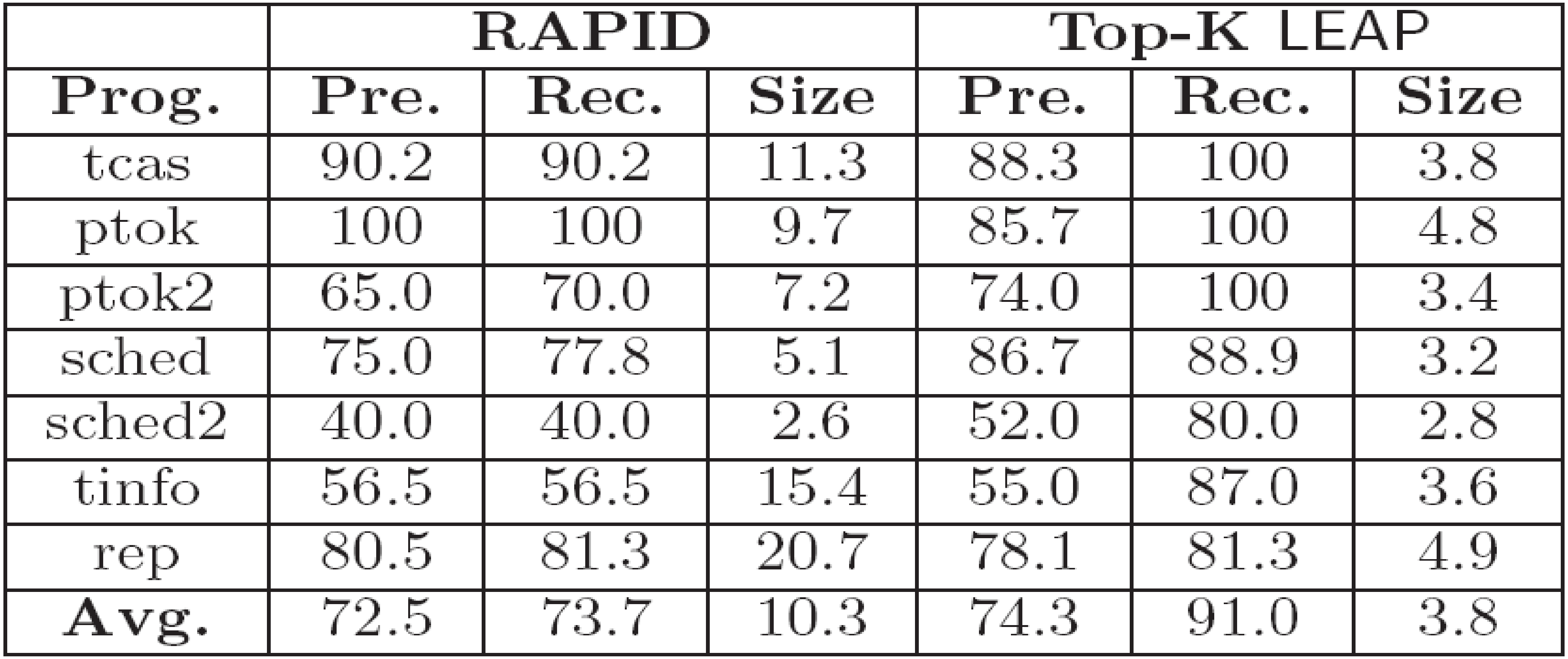
- Datasets
  - Siemens datasets: All 7 programs, all versions
- Methods
  - RAPID [Hsu et al., ASE'08]
  - Top-K LEAP: our method
- Metrics
  - Recall and Precision from top-k returned signatures
  - Recall = proportion of the bugs that could be found by the bug signatures
  - Precision = proportion of the returned results that highlight the bug
  - Distance-based metric to exact bug location penalize the bug context

Experimental Results (Top 5)

	RAPID			Top-K LEAP		
Prog.	**Pre.**	**Rec.**	**Size**	**Pre.**	**Rec.**	**Size**
tcas	82.9	82.9	8.0	85.9	95.1	5.0
ptok	71.4	71.4	4.0	85.7	100	4.3
ptok2	20.0	20.0	2.7	36.0	60.0	2.9
sched	33.3	33.3	2.3	54.1	66.7	3.6
sched2	0.0	0.0	N/A	24.2	30.0	2.2
tinfo	21.7	21.7	2.5	69.6	78.3	2.4
rep	53.1	53.1	5.1	54.4	81.3	2.9
Avg.	40.4	40.4	4.1	58.5	73.0	3.3

Result - Method Level

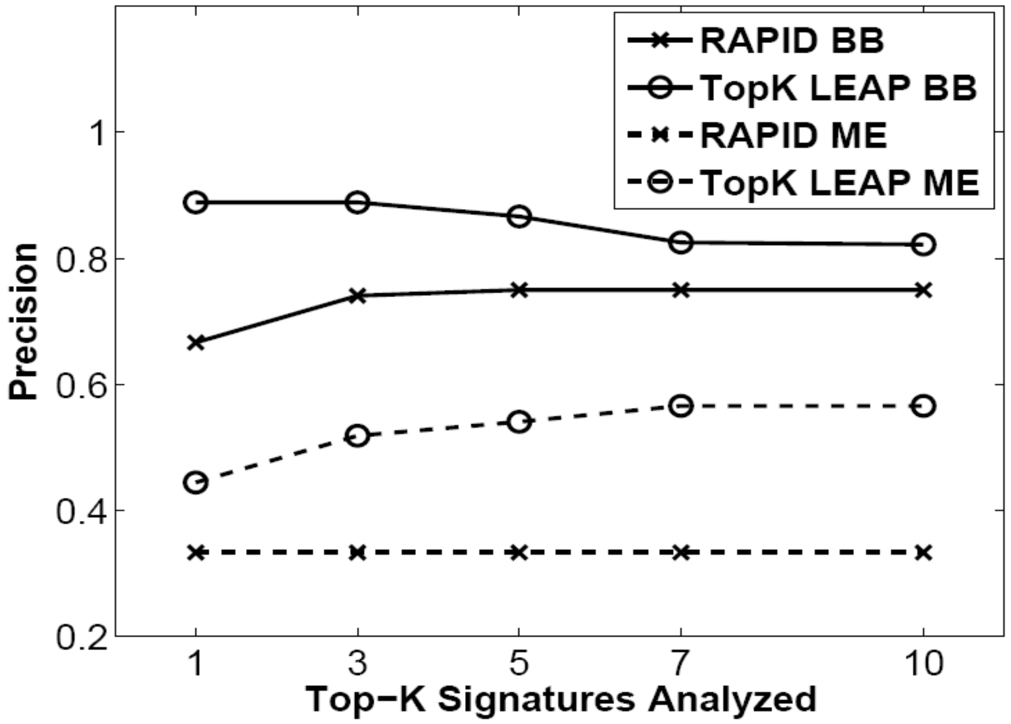
Experimental Results (Top 5)



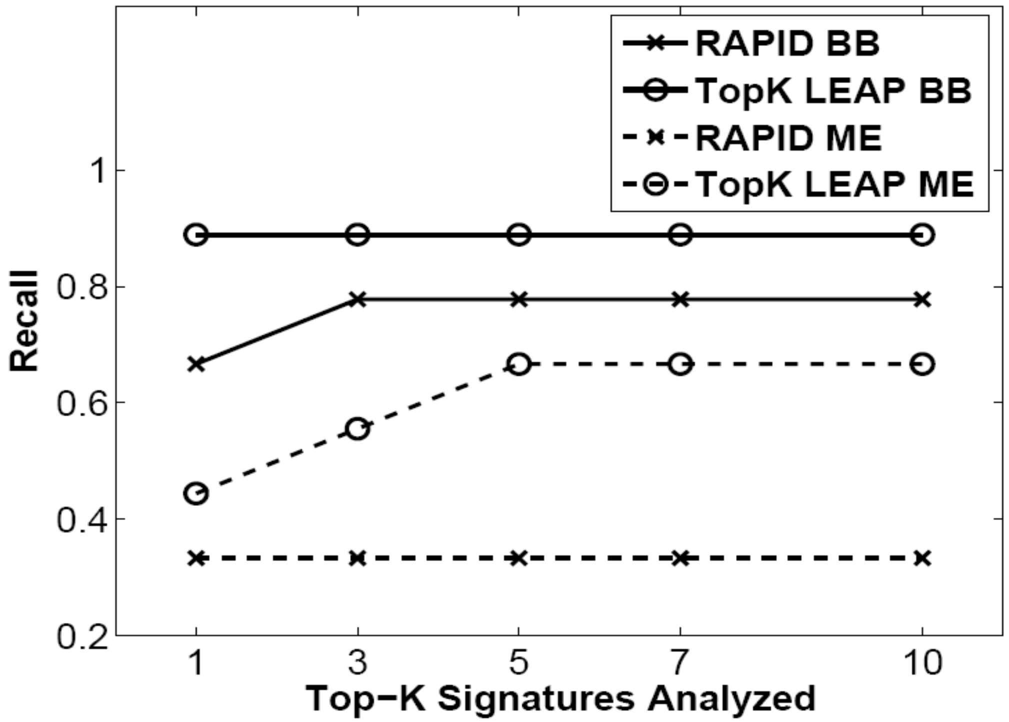
	RAPID			Top-K LEAP		
Prog.	**Pre.**	**Rec.**	**Size**	**Pre.**	**Rec.**	**Size**
tcas	90.2	90.2	11.3	88.3	100	3.8
ptok	100	100	9.7	85.7	100	4.8
ptok2	65.0	70.0	7.2	74.0	100	3.4
sched	75.0	77.8	5.1	86.7	88.9	3.2
sched2	40.0	40.0	2.6	52.0	80.0	2.8
tinfo	56.5	56.5	15.4	55.0	87.0	3.6
rep	80.5	81.3	20.7	78.1	81.3	4.9
Avg.	72.5	73.7	10.3	74.3	91.0	3.8

Result – Basic Block Level

Experimental Results (2) - Schedule



Precision



Recall

Efficiency Test

- Top-K LEAP finishes mining on every dataset between 1 and 258 seconds

- RAPID cannot finish running on several datasets in hours
  - Version 6 of replace dataset, basic block level
  - Version 10 of print_tokens2, basic block level

Experience (1)

```
upgrade_process_prio(prio, ratio){
      ...
      n = (int) (count*ratio+1);
      if(ratio == 1.0) n--; //added code
      proc = find_nth(src_queue, n);
...}

unblock_process(prio, ratio){
       ...
       n = (int) (count*ratio +1);
       if(ratio == 1.0) n--; //added code
       proc = find_nth(src_queue, n);
...}
```

Version 7 of schedule

Top-K LEAP finds the bug, while RAPID fails

Experience (2)

if (rdf <=0 || cdf <= 0)

```
InfoTbl( r, c, f, pdf ){
    rdf = r-1; cdf = c-1; //->basic block 1
    if ( rdf == 0 || cdf == 0 ) { //bug
        info = -3.0; //->basic block 2
        goto ret3;
    }
    N = 0.0; //->basic block 3
    for ( i = 0; i < r; ++i ){
         //->basic block 4,
    }
    //->basic block 5
    if ( N <= 0.0 ){...}
    ...
}
```

For rdf<0, cdf<0
bb1→bb3→bb5

Our method finds a graph connecting block 3 with block 5 with a transition edge

Version 18 of tot_info

Threat to Validity

- Human error during the labeling process
  - Human is the best judge to decide whether a signature is relevant or not.
- Only small programs
  - Scalability on larger programs
- Only c programs
  - Concept of control flow is universal

Related Work

- Bug Signature Mining: RAPID [Hsu et al., ASE'08]
- Bug Predictors to Faulty CF Path [Jiang et al., ASE'07]
  - Clustering similar bug predictors and inferring approximate path connecting similar predictors in CFG.
  - Our work: finding combination of bug predictors that are discriminative. Result guaranteed to be feasible paths.

- Bug Localization Methods
  - Tarantula [Jones and Harrold, ASE'05], WHITHER [Renieris and Reiss, ASE'03], Delta Debugging [Zeller and Hildebrandt, TSE'02], AskIgor [Cleve and Zeller, ICSE'05], Predicate evaluation [Liblit et al., PLDI'03, PLDI'05], Sober [Liu et al., FSE'05], etc.

Related Work on Graph Mining

- Early work
  - SUBDUE [Holder et al., KDD'94], WARMR [Dehaspe et al., KDD'98]
- Apriori-based approach
  - AGM [Inokuchi et al., PKDD'00]
  - FSG [Kuramochi and Karypis, ICDM'01]
- Pattern-growth approach- state-of-the-art
  - gSpan [Yan and Han, ICDM'02]
  - MoFa [Borgelt and Berthold, ICDM'02]
  - FFSM [Huan et al., ICDM'03]
  - Gaston [Nijssen and Kok, KDD'04]

Conclusions

- A discriminative graph mining approach to identify bug signatures
  - –Compactness, Expressiveness, Efficiency
- Experimental results on Siemens datasets
  - –On average, 18.1% higher precision, 32.6% higher recall (method level)
  - –On average, 1.8% higher precision, 17.3% higher recall (basic block level)
  - –Average signature size of 3.3 nodes (vs. 4.1) (method level) or 3.8 nodes (vs 10.3) (basic block level)
  - –Mining at basic block level is more accurate than method level - (74.3%,91%) vs (58.5%,73%)

Future Extensions

- Mine minimal subgraph patterns
  - Current patterns may contain irrelevant nodes and edges for the bug

- Enrich software behavior graph representation
  - Currently only captures program flow semantics
  - May attach additional information to nodes and edges such as program parameters and return values

Thank you for your attention

Questions? Comments? Advice?

hcheng@se.cuhk.edu.hk davidlo@smu.edu.sg

Bug Signature: Discriminative Sub-Graph

- Given graphs labeled as correct or failing
- Find the most discriminative subgraph
- Information gain: $IG(c|g) = H(c) - H(c|g)$

$$H(c) = -\sum_{i \in \{0,1\}} p(c_i) \log p(c_i)$$

$$H(c|g) = -\sum_{i \in \{0,1\}} p(g_i) \sum_{j \in \{0,1\}} p(c_j|g_i) \log p(c_j|g_i)$$

c – class label, g – subgraph

$p(c_1)$ – proportion of faulty traces

$p(g_1)$ – prop. of traces containing the sub-graph

$p(c_1|g_1)$ – proportion of the traces that are faulty given that the graph is exhibited in the trace.

Other Related Work

- Chao et al. Mining Behavior Graphs [SDM'05]
  - Their work detect if a trace is erroneous or not. We find the discriminating signature from two sets of traces.
  - They mine for all closed patterns and then use them as features for the classification of two sets of traces. Our approach directly mine for top-k discriminative graphs.
- Chang et al. Neglected Conditions [ISSTA'07]
  - Their work mine patterns from code rather than traces.
  - Used for bug finding rather than for finding bug signatures.
  - They find frequent graphs, while we find discriminating graphs.

Other Related Work

- Christodorescu et al. Mining Specifications of Malicious Behaviors [FSE'07]
  - Detect only if a graph appear in malware but never in normal.
  - We detect discriminating features, including cases where a graph pattern appear 500 times in faulty, 1 time in normal
  - At times we only have partial information unless we model everything about software systems. Due to this often we do not have a perfectly discriminating feature.