Bug Localization using Classification for Behavior Graph

Fuyao Zhao, fuyaoz@cs.cmu.edu

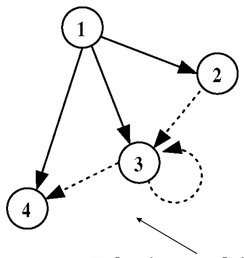
Carnegie Mellon University, School of Computer Science

Problem & Motivation

The size of the program is growing, and the number of bugs is also growing. And debugging become much harder. So we want some algorithms that can automatically help localize bugs for us.

Background

To characterize the program runs, we use behavior graph, which contains a set of nodes represent the function, a set of solid arrows represent function calls and a set of dash arrows represent transitions between each function.



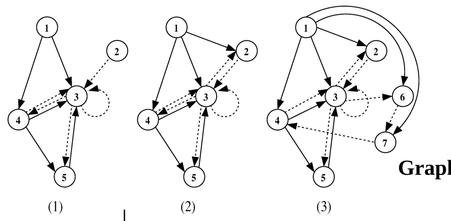
```
int main() {
  scanf("%d", &x);
  y = get(x);
  printf("%d", x);
}
int get(x) {
  if (x > 1)
   return get(x-1)+1;
  return 0;
}
```

Behavior graph if your input is 2

Part I: Graph Classification

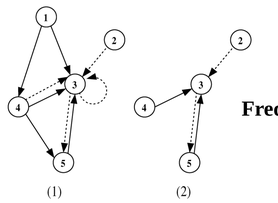
1. Subgraph Extraction

For a set of behavior graph D, we mine closed subgraphs that whose support > threshold. Where support means the frequent that the subgraph appears in D, closed means there is no supergraph has same support.



Graph Set

(1) (2) (3)



Frequent Closed Subgraphs

(1) (2)

Part I: Graph Classification

2. Weighted Graph Model

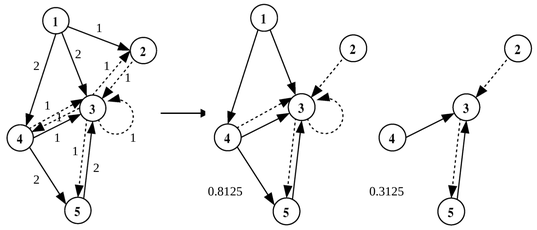
We use weighted model for the behavior graph. For edge $e = (u, v)$, let

$$w(e) = \frac{c_e}{\sum_{e' \in g} w(e')}$$

where c_e is the number of times u calls v. And the weight for a subgraph is

$$w(g') = \sum_{e \in g'} w(e)$$

which indicate how import is the subgraph in the graph. Following example shows the weights for two subgraph in left graph. The edges of left graph are labeled as numbers, which are the call numbers.



3. Graph Classification

We use the subgraphs we mined in 1 as features for data, and calculate the value of each feature use the model in 2. Then, SVM^{light} with linear kernel is used for final classification.

4. Performance Measurement

To measure the performance of the graph classifier, we use 5-fold cross validation. and since the number of incorrect runs in the dataset is very small, accuracy is not a good measurement to evaluate the classifier. So we use F-Score, defined as below:

$$recall = \frac{\#\text{succefully classified incorrect runs}}{\#\text{incorrect runs}}$$

$$precision = \frac{\#\text{succefully classified incorrect runs}}{\#\text{all runs classified as incorrect}}$$

$$\text{F-Score} = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Part II: Bug Localization

1. Generate Behavior Graph

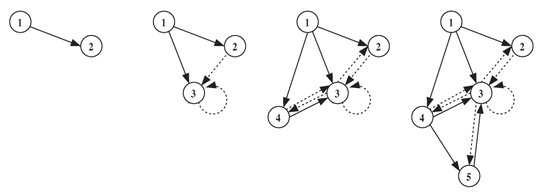
Run the program using different inputs under debug mode, using *gprof* to generate the program run details for behavior graph from the debug information.

2. Checkpoint

Each function F_i has two checkpoints: B^i_{in} and B^i_{out}, correspond to the entrance and the exit of F_i. P^i_{in}, P^i_{out} is the classification performance in B^i_{in}, B^i_{out}. We propose a simplified and more efficient form, which assume

$$P^i_{in} = P^j_{in}, \qquad P^i_{out} = P^k_{out}$$

, where F_j be the first function called by F_i, and F_k be the last called by F_i, to reduce the number of checkpoints. For every program runs, each checkpoint also corresponds to a behavior graph. Following example shows a run that generate 4 graphs for its checkpoints.



3. Graph Set Collection

Now, collect all graphs on same checkpoint generated by different runs. For each set of these graphs, we run graph classification for it to get P^i_{in}, P^i_{out}. And use our assumption to get P^i_{in}, P^i_{out} for all others.

4. Bug Like Function Detection

Define a function F_i is said bug like if

$$P^i_{out} - P^i_{in} > \theta$$

Then a set of bug like function could be found. And we could line them up in a back trace order form on the call graph of an incorrect run, which will hopefully help the programmers to indentify and correct the bugs.

Experiment

1. Dataset:

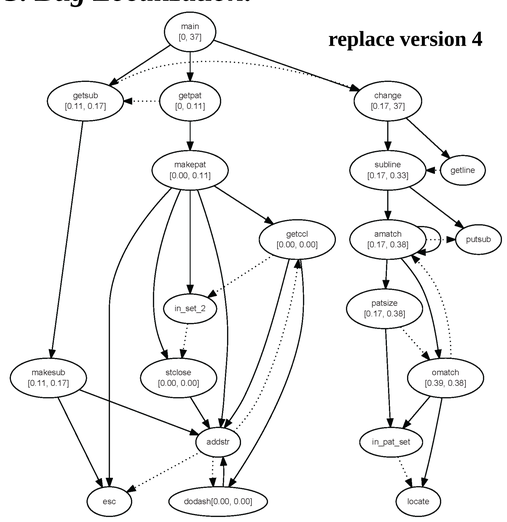
We use the program *replace* in *Siemens Program Set*. It has a standard version and several bug version, (version 1, 2, 3, 4, 5, 6 are used here).

2. Graph Classification:

	Correct	Incorrect	F-Score	Features	Time
version 1	5478	64	0.44	103	1.96s
version 2	5507	35	0.00	103	0.83s
version 3	5414	128	0.37	105	0.22s
version 4	5401	141	0.49	105	0.80s
version 5	5280	262	0.31	105	3.13s
version 6	5459	83	0.17	105	0.16s

3. Bug Localization:

replace version 4



In above behavior graph, we found two sequence of bug like functions:

Bug here!

- ✓ `main > getpat > matpat`
- ✓ `main > change > subline > amatch > patsize > in_pat_set`

Conclusion

- In software testing phrase, it can successfully reduce the work for programmers to find bugs.
- Highly depend on a good graph mining and classification algorithm since the negative example is scarce.