The Journal of Systems and Software 134 (2017) 44–53

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

The Bayesian Network based program dependence graph and its application to fault localization

Xiao Yu[a], Jin Liu[a,*], Zijiang Yang[b,*], Xiao Liu[c]

[a] State Key Laboratory of Software Engineering, Computer School, Wuhan University, Wuhan, China
[b] Department of Computer Science, Western Michigan University, MI, USA
[c] School of Information Technology, Deakin University, Melbourne, Australia

ARTICLE INFO

Article history:
Received 7 March 2017
Revised 4 July 2017
Accepted 12 August 2017
Available online 30 August 2017

Keywords:
Fault localization
Bayesian network
Program analysis

ABSTRACT

Fault localization is an important and expensive task in software debugging. Some probabilistic graphical models such as probabilistic program dependence graph (PPDG) have been used in fault localization. However, PPDG is insufficient to reason across nonadjacent nodes and only support making inference about local anomaly. In this paper, we propose a novel probabilistic graphical model called Bayesian Network based Program Dependence Graph (BNPDG) that has the excellent inference capability for reasoning across nonadjacent nodes. We focus on applying the BNPDG to fault localization. Compared with the PPDG, our BNPDG-based fault localization approach overcomes the reasoning limitation across nonadjacent nodes and provides more precise fault localization by taking its output nodes as the common conditions to calculate the conditional probability of each non-output node. The experimental results show that our BNPDG-based fault localization approach can significantly improve the effectiveness of fault localization.

1. Introduction

As software systems today are larger and more complex than ever before, the potential software defects are increasing (Zhou et al., 2012; Chen et al., 2016). Consequently, debugging, which is one of the most important tasks during software lifecycle, is facing greater challenges. To fix the software defect, one must first be able to locate the fault (Xie et al., 2013a; 2013b). Known as fault localization, this task can be extremely difficult and costly (Xie et al., 2013c; Chen et al., 2015). In recent years, there have been considerable approaches for automated fault localization, including program slicing, statistical models, and probabilistic graphical models.

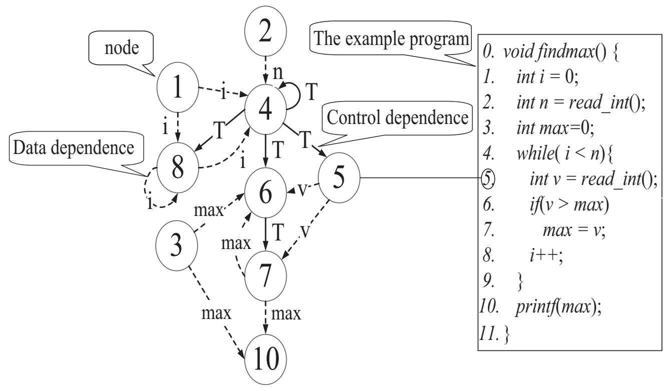
Baah et al. (2010) proposed Probabilistic Program Dependence Graph (PPDG) to capture the conditional statistical dependence and independence relationship among program elements. The ability of probabilistic reasoning about program behaviors make PPDGs valuable for fault localization. However, PPDG is insufficient to reason across nonadjacent nodes and only support making inference about the local anomaly, and the reason for this is twofold. First, PPDG is based on the dependency network that does not support bidirectional reasoning between two adjacent nodes and transitive reasoning that compare two nodes via intermediate network nodes. Second, PPDG assumes that the fault nodes can be located according to the comparison of the conditional probabilities of nodes given the states of their parent nodes, which reflects how the parent nodes influence their children nodes. This assumption is valid when faults occur just between child nodes and their adjacent parent nodes. But the validation of this assumption cannot be guaranteed when faults occur across nonadjacent nodes.

In this paper, we propose a novel probabilistic graphical model called Bayesian Network based Program Dependence Graph (BNPDG). Our technique produces the BNPDG for a program by augmenting its program dependence graph automatically. Each node in BNPNG is associated with a conditional probability table (CPT) that indicates the conditional probability distribution. CPT relates the state of the node to the state of its parents. Since a PDG containing cycles violates the acyclic assumption of Bayesian network (BN), we apply the short-cycle-first heuristic and the Maximal Information Coefficient (MIC) to eliminate cycles. Compared to PPDG, BNPDG is able to reason across nonadjacent nodes, because it is a directed acyclic graph with bidirectional inference ability. That is, BNPDG can calculate the conditional probability of a node X given the states of any nonadjacent nodes.

Since the reasoning ability of BNPDG makes it possible to perform more challenging tasks such as fault localization in fault diagnosis, we focus on applying the BNPDG to fault localization. PPDG

* Corresponding author.
E-mail addresses: jinliu@whu.edu.cn, mailjinliu@yahoo.com (J. Liu), zijiang.yang@wmich.edu (Z. Yang).

http://dx.doi.org/10.1016/j.jss.2017.08.025
0164-1212/© 2017 Elsevier Inc. All rights reserved.



```
0.  void findmax() {
1.    int i = 0;
2.    int n = read_int();
3.    int max=0;
4.    while( i < n){
5.      int v = read_int();
6.      if(v > max)
7.        max = v;
8.      i++;
9.    }
10.   printf(max);
11. }
```

Fig. 1. A typical example of the PDG that corresponds to the program "*findmax*".

(Baah et al., 2010) assumes that the faulty nodes can be located based on the comparison of the conditional probabilities of the node given the state of their parents, which is valid only when faults occur just between the children nodes and their adjacent parent nodes. Compared with the PPDG, the BNPDG overcomes this reasoning limitation and provides more precise fault localization by taking its output nodes as the common conditions to calculate the conditional probability of each non-output node.

To summarize, we mainly make the following contributions:

(1) We propose a Bayesian network based probabilistic graphical model that has better reasoning capability than its rivals.
(2) We propose a BNPDG-based fault localization approach.
(3) We experimentally evaluate our approach and five other existing fault localization methods on publically available benchmarks that include SOBER, Tarantula, and RankCP. The experimental results show that our approach is more accurate and more scalable.

The remainder of the paper is organized as follows. Section 2 presents necessary background on models on which the BNPDG is based. Section 3 presents the details of the construction of the BNPDG. Section 4 explains our BNPDG-based fault localization approach and Section 5 conducts empirical studies to evaluate the performance. Section 6 discusses the potential threats to validity and Section 7 describes related work. Finally, Section 8 concludes the paper with a discussion of the future work.

2. Backgound

In this section, we briefly review the three models that form the basis for the BNPDG. The first is the program dependence graph (PDG) that abstracts the dependence relationship between statements into a graph. The second is the Bayesian network (BN) that represents a joint probability distribution over a set of stochastic variables. The third is the Maximal Information Coefficient (MIC) that measures the correlation between two variables.

2.1. Program dependence graph

PDG abstracts the dependence relationship of a program into a graph, where each node denotes a statement and each edge presents the control or data dependence between statements (Ferrante et al., 1987; Hammer and Snelting; 2009). Fig. 1 shows the PDG for program *findmax* (Baah et al., 2010) that finds the maximum number from a set of integers. The solid and dashed edges present the control and the data dependence, respectively. For example, in Fig. 1, node 6 is control-dependent on node 4 and data-dependent on nodes 3, 5, and 7. A control dependence edge may be further labeled with "T" that indicates the execution is taken along the edge if the condition is true. The data dependence edges are labeled with the variables on which there exists data dependence. For example, the data dependence edge between node 3 and node 5 has the label "max", which indicates that the value of variable max at node 3 flows to node 6.

2.2. Bayesian Network

BN characterizes uncertain knowledge based on the probability theory and the graph theory. It can be used to represent the causal information and discover the potential variable relationship (Neapolitan, 2003; Peng and Ding, 2003). The construction of a BN consists of the procedures of structure learning and parameter estimation. Structure learning elucidates the structure of BN. Parameter estimation calculates the Conditional Probability Tables (CPTs) of a BN. For each node in a BN, it is necessary to estimate its conditional probability given their parents. Taking advantage of the PDG, our approach applies expert knowledge to structure learning in constructing a BN. During parameter estimation, we establish the CPTs by analyzing the frequency of the samples. The most important ability of a BN is the Bayesian inference that supports the process of answering the queries. A query represented as $p(Y|E=e)=\frac{P(Y,e)}{P(e)}=\frac{P(e|Y)P(Y)}{P(e)}$ indicates the posterior probability distribution of the variable Y given the condition of the variable E=e. Many algorithms implement the Bayesian inference. Our approach adopts junction tree propagation (Peng and Ding, 2003) to query the posterior probability distribution in fault location.

2.3. Maximal information coefficient

The maximal information coefficient (MIC) measures dependence between two variables and quantifies the relevance based on large datasets (Reshef et al., 2011). Compared with other statistics, MIC facilitates discovering various types of relationships such as the linear function, the nonlinear function and the non-functional relationship (Reshef et al., 2011). MIC is based on the theory of mutual information; hence we briefly describe the theory before introducing MIC.

Let X be a random variable with discrete values. The entropy of X is defined as

$$H(\mathrm{X}) = -\sum_{x\in X} p(x)\log\ p(x) \tag{1}$$

where $p(x)$ is the probability density function of X. The joint entropy $H(X,Y)$ of two the random variables X and Y is defined as

$$H(\mathrm{X},\mathrm{Y}) = -\sum_{y\in Y}\sum_{x\in X} p(x,y)\log\ p(x,y) \tag{2}$$

To quantify the reduction in uncertainty about variable X (Y after observing variable Y, or by symmetry, the reduction in uncertainty about Y after observing X, the mutual information is defined as follows.

$$I(\mathrm{X};\mathrm{Y}) = H(\mathrm{X}) + H(\mathrm{Y}) - H(\mathrm{X},\mathrm{Y}) \tag{3}$$

$$i.e., I(\mathrm{X};\mathrm{Y}) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} \tag{4}$$

Mutual information is a measure of dependence. Based on the above formula, the value of $I(X;Y)$ is 0 if variables X and Y are independent; otherwise, the value is greater than zero. The greater the value is, the more relevant the two variables are. MIC is designed by the following principle: if there exists a correlation between two variables X and Y, a grid can be drawn on the scatter diagram of the two variables to make most of the data points fall into several cells of the grid. By searching for the optimal grid, MIC calculates correlation of two variables by counting the cells.

Given a finite dataset D, let X and Y be two variables with a sample size n. Suppose that the values of the two variables are divided into x bins and y bins, respectively, we call this partition as an x-by-y grid. Let $D|_G$ denote the distribution of the points in D on the cells of a grid G. For each cell of G, the probability mass of the cell is the percentage of points falling into the cell. Thus for different x-by-y partitions, we can obtain different distributions of $D|_G$.

For a specific x-by-y partition, the maximum mutual information of $D|_G$ is defined as

$$I^*(D, x, y) = \max I(D|_G) \tag{5}$$

where $I(D|_G)$ denotes the mutual information of $D|_G$. That is, $I^*(D, x, y)$ is the maximum value of $I(D|_G)$ for all cells of the grid.

For different x-by-y partitions, we can obtain different values of $I^*(D, x, y)$. Then, under different x-by-y partitions, a characteristic matrix $M(D)$ can be constructed by choosing $I^*(D, x, y)$ of each x-by-y partition as

$$M(D_{x,y}) = \frac{I^*(D, x, y)}{\log(\min\{x, y\})} \tag{6}$$

where normalizing by $\log(min\{x, y\})$ can make the entries of the matrix range from zero to one and guarantee that all noiseless functions get perfect mutual information scores. Furthermore, the MIC value can be defined as

$$\mathrm{MIC}(D_{x,y}) = \max_{xy<B(n)} \{M(D_{x,y})\} \tag{7}$$

where $B(n)$ is the upper bound of the grid size. In this paper, we follow (Reshef et al., 2011) to set $B(n) = n^{0.6}$ as the default value.

3. Bayesian Network based program dependence graph

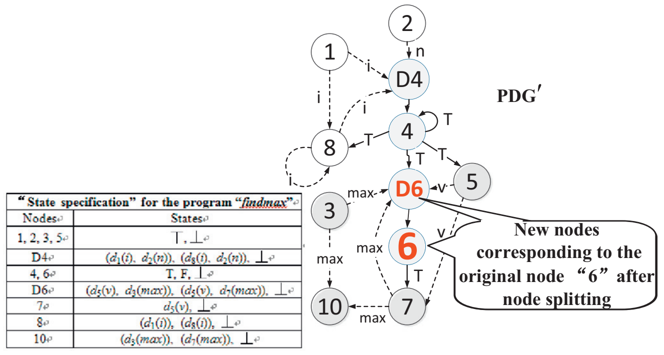
BNPDG can be produced by transforming the PDG of a program into a BN. There are two reasons that we choose BN instead of other probabilistic graphical models to establish our target model. First, BN is a directed graph, which is more suitable than other undirected probabilistic graphical models to represent the directed control and data dependencies between nodes in the PDGs. Second, BN is acyclic, which has stronger inference ability than directed dependency network with cycles.

Definition 1. The Bayesian Network based Program Dependence Graph (BNPDG) for program P is a triple (G, S, Q), where $G = (N, E)$ is the transformed PDG of P. N represents the program components of the program P and E represents the control and data dependence in P. S is the mapping from nodes to states, and Q is the mapping from nodes to conditional probability distributions.

The construction of the BNPDG contains of three steps: node splitting and state specification, cycle elimination and parameter estimation. We present an in-depth discussion of these steps below.

3.1. Node splitting and state specification

Our approach takes each node in the BNPDG as a random variable that corresponds to a set of mutually exclusive states. All states of a node describe various cases for this node when the program executes. A node in a PDG is a predicate node when it represents a branch predicate. A node is a non-predicate node when it represents a program statement that uses one or more variables. Before specifying the state for a PDG node, it is necessary to clarify the type of this node because the ways that specify the state for a predicate node and a non-predicate node are different. Moreover,



PDG′
New nodes corresponding to the original node "6" after node splitting
1
2
D4
4
8
5
3
D6
6
10
7
n
i
T
max
v

"State specification" for the program "*findmax*"	
Nodes	States
1, 2, 3, 5	$\top$, $\perp$
D4	$(d_1(i), d_2(n))$, $(d_8(i), d_2(n))$, $\perp$
4, 6	T, F, $\perp$
D6	$(d_5(v), d_3(max))$, $(d_5(v), d_7(max))$, $\perp$
7	$d_5(v)$, $\perp$
8	$(d_1(i))$, $(d_8(i))$, $\perp$
10	$(d_3(max))$, $(d_7(max))$, $\perp$

Fig. 2. Operation result of node splitting and state specification for the program *findmax.*

for a node contains two state components, our approach splits it into a predicate node that represents the predicate component and a non-predicate node that represents the component of data dependence. For the immediate predecessor of the split node, the newly generated non-predicate node will be its immediate successor. For the immediate successors of the split node, the newly generated predicate node will be its immediate predecessor. Thus, the newly generated non-predicate node becomes the immediate predecessor of the newly generated predicate node. As a result, the operation of node splitting transforms PDG into PDG'.

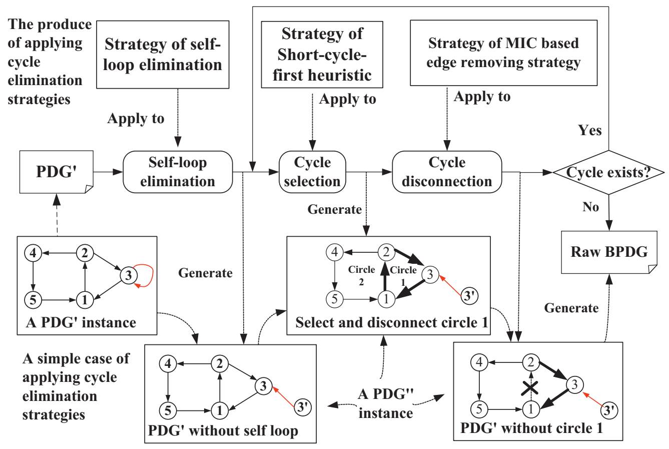
Our approach uses the state set {T, F, $\perp$} to represent the states of a predicate component, where "T" and "F" represent the predicate outcomes, and "$\perp$" means that the node was not executed. To specify the state of a non-predicate component, our approach takes the context of this non-predicate component into consideration. The state set of a non-predicate component denotes the place where the data come from for each variable contained in the statement (Laski and Korel, 1983).

Fig. 2 illustrates the operation result of node splitting and state specification for the program "*findmax*". In particular, since node 6 corresponds to the statement "*if* (v > max)", the original node 6 in Fig. 2 was split into the new node 6 for its predicate component and the new node D6 for its non-predicate component. As a predicate component, the new node 6 was specified with the state set {T, F, $\perp$}, where "$\perp$" means the node was not executed. Oppositely, "$\top$" means that the node was executed. As a non-predicate component, the new node D6 contains two variables v and max in the statement "*if* (v > max)". The variable max has been defined at node 3 and node 7. But only one of these definitions can be used by node D6. This mutually exclusive situation of node 3 and node 7 can be denoted as d_3(max) and d_7(max) respectively. Similarly, the variable v was denoted as $d_5(v)$. As a result, the state set of D6 can be specified as {($d_5(v)$, d_3(max)), ($d_5(v)$, d_7(max)), $\perp$}.

3.2. Cycle elimination

Since BN is a directed acyclic graph, we have to eliminate cycles in a PDG'. Our approach exploit three strategies: the self-loop elimination strategy, the short-cycle-first heuristic strategy and the MIC based edge removing strategy.

First, self-loops in a PDG′ are eliminated with the self-loop elimination strategy. To distinguish from the PDG′, the PDG after the self-loop elimination is denoted as PDG′′. Then, the short-cycle-first heuristic strategy is applied to choose a cycle prior to be disconnected from multi cycles that may be found in the PDG′′. At last, our approach applies the MIC based edge removing strategy to eliminate the selected cycle in the graph by disconnecting edges from this cycle. After that, our approach reiterates the short-cycle-first heuristic strategy and the MIC based edge removing strategy



The produce of applying cycle elimination strategies
Strategy of self-loop elimination
Strategy of Short-cycle-first heuristic
Strategy of MIC based edge removing strategy
Apply to
Apply to
Apply to
PDG'
Self-loop elimination
Cycle selection
Cycle disconnection
Cycle exists?
Yes
No
Raw BPDG
Generate
Generate
Generate
A PDG' instance
Select and disconnect circle 1
Circle 2
Circle 1
A simple case of applying cycle elimination strategies
PDG' without self loop
A PDG'' instance
PDG' without circle 1

Fig. 3. Procedure of applying cycle elimination strategies.

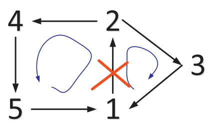


Fig. 4. A case applying short-cycle-first heuristic strategy.

until all cycles were removed from the PDG″. The procedure of applying cycle elimination strategies with a transformation case PDG″ is presented in Fig. 3.

3.2.1. Self-loop elimination strategy

The self-loop elimination strategy eliminates a self-loop by removing this self-loop and adds a new node. The new node is an immediate predecessor of the original node and is data dependent or control dependent on the original node. For example, there is a cycle at node 3 in the PDG′ in Fig. 3 because the node 3 is data-dependent on itself. According to the self-loop elimination strategy, this cycle was removed by adding the node 3′ as the predecessor of the node 3. A newly added node is assigned with the same state as its corresponding original node. That is, node 3′ keeps the same state set as node 3.

3.2.2. Short-cycle-first heuristic strategy

After self-loop elimination, there may be still cycles in PDG″. In this case, PDG″ identifies the cycles to be disconnected according to the short-cycle-first heuristic strategy that detect the cycle with the minimum length (Peng and Ding, 2003).

Because information propagates multiplicatively within the BNs according to the probability product rule, the influence of a node on the other along a fixed path can be calculated as $P_1 \cdot P_2 \ldots \cdot P_m$ approximately if the length of the path is m. Accordingly, the shorter a cycle is, the more severely the cycle breaks the acyclic restriction for the BN.

The short-cycle-first heuristic strategy is more efficient at cycle elimination than other strategies. When cycles on a graph share the common edges, the short-cycle-first heuristic strategy tends to disconnect a cycle with the shortest cycle length firstly by removing a common edge. Such approach is more likely to disconnect other cycles due to the removal of their common edges.

For example, the graph in Fig. 4 contains two cycles that shares an edge "$1 \to 2$", i.e., "$1 \to 2 \to 3 \to 1$" with the shortest length 3 and "$1 \to 2 \to 4 \to 5 \to 1$" with the length 4. While the cycle "$1 \to 2 \to 3 \to 1$" would remove the common edge "$1 \to 2$" at the probability of 1/3, the cycle "$1 \to 2 \to 4 \to 5 \to 1$" would remove the common edge "$1 \to 2$" at the probability of 1/4. Thus, the short cycle has a higher probability than the long cycle in removing the common edge. Moreover, the removal of the common edge may also disconnect other cycles sharing this common edge, e.g., the disconnection of the cycle "$1 \to 2 \to 4 \to 5 \to 1$" due to removal of the edge "$1 \to 2$" from the cycle "$1 \to 2 \to 3 \to 1$".

Algorithm 1 Cycle_Elimination.

```
   input: PDG′
   output: Raw BNPDG
1  if PDG′ contains self-loops then
2      Get PDG″ from PDG′ with the self-loops elimination strategy;
3  else
4      PDG″ ←PDG′
5  end
6  while cycles in PDG″ do
7      Select $C_{shortest}(C_1 \to C_2 \to \ldots \to C_m \to C_1)$ from PDG″ with the short-cycle-first heuristic strategy
8      //Disconnect $C_{shortest}$ in G′ according to the MIC based edge removing strategy
9      Select MIC($C_i$,$C_{i+1}$)=
           min{MIC($C_1$,$C_2$),MIC($C_2$,$C_3$),…,MIC($C_{m-1}$,$C_m$),MIC($C_m$,$C_1$)}
10     //Disconnect $C_{shortest}$
11     Remove edge "$C_i \to C_{i+1}$" from G′
12 end
13 return G′
```

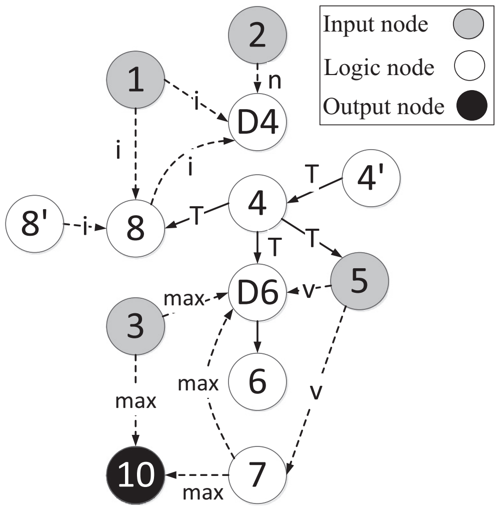


Fig. 5. A raw BNPDG after cycle elimination with our strategies for the program *findmax*.

3.2.3. MIC based edge removing strategy

The elimination of cycle not only satisfies the structural definition of the Bayesian network, but also maintains the completeness of the dependency information in a PDG as much as possible. Thus, our approach disconnects a cycle by removing an edge with the smallest probability of the dependent relationship, which is measured by the statistic MIC according to Eq. 7. MIC is better than other relation measuring statistics in generality and equitability. Generality means that the ability to detect a wide range of relationships, not limited to functional types such as linear, exponential, or periodic, or even to nonfunctional types. Equitability refers to the ability to overcome the deviation made by the preferences of statistics to special types of relationships. Accordingly, the higher the MIC value is the more possible a strong dependence exists between two variables. Otherwise, it is impossible for variables to have dependent relationships.

Algorithm 1 presents the pseudo-code that transforms a PDG into a BNPDG by using the aforementioned three strategies to eliminate cycles. For the benchmark case of the PDG′ in Section 3.1, Fig. 5 illustrates the result after cycle elimination with our strategies for the program *findmax*. The end result is a raw BNPDG for further processing in the parameter estimation phase.

According to Algorithm 1, the PDG′ turns into a PDG′′ after removing two self-loops "4 → 4" and "8 → 8". Two nodes "4" and "8′" are added to the PDG′′ according to the self-loop elimination strategy. But the PDG′′ still contains two cycles "4 → 8 → D4 → 4" and "6 → 7 → D6 → 6". By the short-cycle-first heuristic strategy and the MIC based edge removing strategy, two edges "D4 → 4" with MIC value 0.03169 and "6 → 7" with MIC value 0.04828 in our experiment are removed from the PDG′′, respectively.

3.3. Parameter estimation

After structure learning identifies the network structure of the raw BNPDG, the execution of the testing program covers various execution instances and generates the abundant "execution data" for parameter estimation, denoted as *D*. Our technology estimates the parameters of the BNPDG by using the execution instances of passing test cases, which enables the BNPDG to capture the correct behaviors of the program. After building the BNPDG, we use the execution instances of failing test cases to locate the fault.

Parameter estimation establishes the BNPDG by estimating the conditional probability of each node given its parents through *D*, which are represented as tables called conditional probability tables (CPTs). For the BNPDG with *m* nodes and the target program executing *n* times, the corresponding "execution data" can be represented as $D = \{d_1, d_2, \ldots, d_n\}$, where d_i is the observed result in the *i*th execution. d_i can further be described as a trace of node-state pairs $d_i = \{(X_1\colon x_{1i}), (X_2\colon x_{2i}), \ldots, (X_k\colon x_{ki}), \ldots, (X_m\colon x_{mi})\}$, where X_k represents the *k*th node and x_{ki} represents the execution state of the *k*th node in the *i*th execution. A node X_k may or may not have parents. For X_k without parents, the probability that the state of X_k is x_{kj} can be estimated according to formula 8:

$$p\left(X_k = x_{kj}\right) \approx \frac{n\left(X_k = x_{kj}\right)}{n(X_k)} \quad (j = 1, 2, \ldots, s) \tag{8}$$

where x_{kj} represents the *j*th state of the node X_k, $n(X_k)$ represents the total number of execution instances of the target program, and $n(X_k = x_{kj})$ represents the number of execution instances that satisfy the constrain "the state of X_k is x_{kj}".

For X_k with parents $Pa(X_k)$, the probability that the state of X_k is x_{kj} given the state of $Pa(X_k)$ is pa_{kj} can be estimated according to the following formula:

$$p\left(X_k = x_{kj} | Pa(X_k) = pa_{kj}\right) \approx \frac{n(X_k = x_{kj}, Pa(X_k) = pa_{kj})}{n(Pa(X_k) = pa_{kj})} \tag{9}$$

where pa_{kj} represents the *j*th state combination of $Pa(X_k)$, $n(Pa(X_k) = pa_{kj})$ represents the number of execution instances that satisfy the constrain "the state combination of $Pa(X_k)$ is pa_{kj}", and $n(X_k = x_{kj}, Pa(X_k) = pakj)$ represents the number of instances that satisfy the constrain "the state of X_k is x_{kj} and the state combination of $Pa(X_k)$ is pa_{kj}.

Take the estimation of the CPT for node 6 in Fig. 5 for example, since D6 is the parent node of node 6, the conditional probability for node 6 is represented by P(6|D6). Therefore, we need to estimate the conditional probabilities of states of node 6 given the states of node D6. Table 1 shows the CPT for node 6. The first column shows the states of node D6 and the second row shows the states of node 6. The table shows that P(6: T | D6: $(d_5(v), d_3(\max))$) = 0.97, which means that the conditional probability of node 6 with the state "T" is 0.97 given the node D6 with the state "$(d_5(v), d_3(\max))$".

Table 1
CPT of node 6.

node D6	node 6		
	T	F	⊥
$(d_5(v), d_3(\max))$	0.97	0.03	0.00
$(d_5(v), d_7(\max))$	0.49	0.51	0.00
⊥	0.00	0.00	0.00

4. Fualt localization with the BNPDG

The reasoning ability of the BNPDG makes it possible to perform tasks such as fault localization. In general, the categories of program errors are syntax errors, semantic errors and logic errors (Hristova et al., 2003). Our approach mainly focuses on logic errors that cause the program to operate incorrectly. A logic error tends to produce incorrect outputs.

Our approach categorizes the nodes in the BNPDG into input nodes, output nodes and logic nodes. Input nodes represent the initialization statements in the program, e.g., nodes 1, 2, 3 and 5 in Fig. 5. Output nodes represent the result of the program and the output statement, e.g., node 10 in Fig. 5. Logic nodes refer to other nodes that are not input or output nodes in an BNPDG, e.g., nodes 4, 4′, D4, 6, D6, 7, 7′, 8 and 8′ in Fig. 5.

The PPDG based fault localization approach (RankCP) (Baah et al., 2010) also judges the fault probability of a node based on the conditional probability of this node given the state of their parent nodes. This is achieved by using $p(X_j = x_{ji} | Pa(X_j) = pa_{ji})$ that reflects how the parent nodes influence their children nodes. The hypothesis is that a node X_j with an unusual parent state is a possible cause of the execution failure (Baah et al., 2010). But the validation of this assumption cannot be guaranteed when faults occur across nonadjacent nodes.

The advantage of BNPDG-based fault localization approach over the PPDG-based approach is that the former takes the global effect of the fault into consideration and synthesizes the state of all output nodes to represent the global result of the program execution. The BNPDG-based fault localization approach takes the output nodes as the common condition to calculate the conditional probability of each non-output node given the state of the output nodes, which leads to reason across nonadjacent nodes.

It detects the potential fault by ranking logic nodes with their conditional probabilities given the state of the output nodes in a descending order. If the state of each logic node X_j in the BNPDG is x_{ji}, the conditional probability of the logic node can be calculated with the Bayesian inference by the following formula:

$$p\left(X_j = x_{ji} | evidence\right) \tag{10}$$

where *evidence* represents the execution state of all erroneous output nodes. As mentioned in Section 2.2, our approach adopts junction tree propagation (Peng and Ding, 2003) to make Bayesian inference in fault location. Therefore, the hypothesis of the BNPDG-based fault localization approach is that if a logic node with a higher conditional probability of causing the erroneous output, this node is more likely to be the place where a potential fault leads to the failure.

In some cases, a certain kind of logic nodes may be found in the BNPDG with a state whose probability is high no matter what states of the output nodes are, although there may be no faults on these nodes. To deal with this exceptional case, our fault localization approach synthesizes the output nodes into a multiple random variable *SumO* that describes the comprehensive situation of all output nodes. For example, if there are two output nodes *A* and *B* in the BNPDG with two state set $A\{a_1, a_2\}$ and $B\{b_1, b_2\}$ respectively, the *SumO* for nodes *A* and *B* corresponds to a comprehensive state set $\{(A = a_1, B = b_1), (A = a_1, B = b_2), (A = a_2,$

Table 2
Details of experiment dataset.

Prototype program		Faulty versions	LOC	Description
Siemens suite	print_tokens	7	472	Lexical analyzer
	print_tokens2	10	399	Lexical analyzer
	replace	32	512	Pattern replace
	schedule	9	292	Priority schedule
	schedule2	10	301	Priority schedule
	tcas	41	141	Altitude separation
	tot-info	23	440	Information measure
Space		38	6199	Interpreter for ADL

Algorithm 2 Fault localizaiton.

```
   input: $\{X_j : x_{ji}\}_{j=1}^{n}$: a trace of node-state pair
       the BNPDG
       $\xi$: threshold.
   output: Ranked nodes where potential faults exist
1  //Select the logic and output nodes from BNPDG
2  logicNodes $\leftarrow \{L_1 : l_{1k}, L_2 : l_{2k}, \ldots, L_a : l_{ak}\},\ L_j : l_{jk} \in \{X_j : x_{ji}\}_{j=1}^{n}$
3  outputNodes$\leftarrow\{O_1 : o_{1k}, O_2 : o_{2k}, \ldots, O_b : o_{bk}\}, O_j : o_{jk} \in \{X_j : x_{ji}\}_{j=1}^{n}$
4  sumO$\leftarrow(O_1{:}o_{1k}, O_2{:}o_{2k}, \ldots, O_b{:}o_{bk}), O_j{:}o_{jk} \in$ outputNodes;
5    //Exclude the non-suspicious nodes from logicNodes according to MIC
6  for $j=1$ to $a$ do
7    if MIC($L_j$, sumO) $< \xi'$ then
8      logicNodes $\leftarrow$ logicNodes – $\{L_j{:}L_{jk}\}$;
9    end
10 end
11 //Calculate the conditional probability with Bayesian inference
12 for every node within logicNodes $L_i{:}l_{ik}$ do
13   $prob_i = p(L_i = l_{ik} | O_1 = o_{1k},\ O_2 = o_{2k},\ \ldots,\ O_b = o_{bk})$;
14 end
15 Rank logicNodes with $prob_i$ in descending order;
16 return ranked logicNodes
```

$B = b_1$), ($A = a_2$, $B = b_2$)}. Before the calculation and rank of conditional probability for the logic node, the correlation between the logic nodes in the BNPDG and *SumO* will be examined. If the examination gets a lower score than a certain threshold, there is a weak correlation between a logic node and *SumO* so that this node was judged as a correct node and removed from the suspicious nodes.

Our BNPDG-based fault localization approach adopts the statistic MIC to measure the correlations between logic nodes and *SumO*. Our approach of taking MIC to exclude the valid nodes from suspicious logic nodes has at least two advantages. First, it avoids the misjudgment of these special "correct" nodes with a state over a certain high probability. Second, it effectively reduces the search space of ranking the conditional probability for the suspicious logic nodes with Bayesian inference. Our BNPDG-based fault localization approach is presented in Algorithm 2.

To illustrate our BNPDG-based fault localization approach and compare it with RankCP (Baah et al., 2010), we provide an example of how our approach and RankCP are used to locate the fault in a failing execution. We again use the example program "*findmax*", described in Section 2, to illustrate our approach and RankCP. This program contains a logic error at line 3: The initial value of max should be set to the least negative integer value. The program fails when all integers in the input are negative. When the program "*findmax*" receives the failing input (n = 1,v = {-1}), it outputs zero as the maximum value. RankCP ranks a node X_j that has a state whose probability is low, given the states of X_j's parents, as highly suspicious. Therefore, RankCP flags node 6 as the most suspicious node in the trace because it had the least frequent state configuration, i.e., P(6: F|D6: $(d_5(v), d_3(\max))$) = 0.03 as shown in Table 1. However, the faulty location is at node 3. For this failing input, RankCP does not pinpoint the faulty location. The reason why node 6 is ranked the highest is that RankCP assumes that the fault nodes can be located according to the comparison of the conditional probabilities of nodes given the states of their parent nodes, which reflect how the parent nodes influence their children nodes. This assumption is valid when faults occur just between child nodes and their adjacent parent nodes. But the validation of this assumption cannot be guaranteed when faults occur across nonadjacent nodes. In this example, node 6 and the faulty node 3 are nonadjacent nodes, therefore, RankCP fails to pinpoint the faulty location.

By contrast, our BNPDG-based fault localization approach locates faults by calculating the conditional probability of logic nodes given the states of the output nodes. That is, our approach calculates the conditional probability of all logic nodes given the condition of the output node 10 by the Bayesian inference. Our approach adopts junction tree propagation (Peng and Ding, 2003) to implement the Bayesian inference. In this example, the corresponding conditional probability for node 6 is $\mathrm{P}(6{:}\ \mathrm{F}|10{:}\ d_3(\max)) = \frac{\mathrm{P}(6{:}\ \mathrm{F},\ 10{:}d3(max))}{\mathrm{P}(10{:}d3(max))} = 0.12$. The node 3 has the highest conditional probability among all logic nodes, which is $\mathrm{P}(3{:}\top|10{:}\ d_3(\max)) = \frac{\mathrm{P}(3{:}\mathrm{T}, 10{:}d_3(max))}{\mathrm{P}(10{:}d_3(max))} = 0.48$. Therefore, our approach flags node 3 as the most suspicious node.

5. Evaluation

In this section, we evaluate our approach for fault localization. We first describe the experimental dataset and the performance measurement. Then, we present the experimental results. Experiments were conducted on a workstation with an Intel Core i7-4790 CPU with 3.60 GHz. We calculate MIC values of two nodes via the MINE toolkit (http://www.exploredata.net/) and use BNT toolkit (https://github.com/bayesnet/bnt) for Bayesian inference in fault localization.

5.1. Data set

In this experiment, we employ the *Siemens suite* and *Space* as our benchmarks, because these programs have been widely used in fault localization. The *Siemens* programs are written in C, and they are a suite of seven small programs, including print_tokens, print_tokens2, replace, schedule, schedule2, tcas and tot-info. Since *Siemens* programs are a suite of seven small programs, we also use the *Space* program for our scalability study. The *Space* program is a software subject developed by the European Space Agency. For our experiments, we intentionally omit 13 versions from the *Siemens suite* and remove 12 versions from the *Space* program. We eliminated these versions because 1) there were no syntactic differences between the C file of the correct version and the faulty versions of the program (e.g., change in header file), 2) no traces could be gathered because the faulty versions had segmentation faults when executed on their test suite, or 3) none of the test cases failed when executed on the faulty version of the program. After removing these versions, 123 versions of *Siemens* programs and 26 versions of *Space* program are left to evaluate the effectiveness of our approach.

5.2. Performance measures

In the experiment, we employ one commonly used performance measurement *Score* that is defined as follows:

$$Score = \frac{|N|}{|sumN|} \times 100 \tag{11}$$

where $|N|$ is the number of statements examined until the faulty node is found according to the ordinal suspicious degree, and $|sumN|$ is the total number of statements. The lower value of *Score* an approach of fault localization receives, the more effective it is.

5.3. Methods in comparison

In order to confirm whether our approach can outperform others, we compare our approach with other approaches that include Tarantula (Jones and Harrold, 2005), SOBER (Liu and Han, 2006; Liu et al., 2005) and CT (Cause Transitions) (Cleve and Zeller, 2005), and RankCP (Baah et al., 2010).

Tarantula is a fault localization approach proposed by Jones and Harrold (2005). The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. In particular, the suspiciousness of a statement, *s*, is computed by the following equation:

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} \tag{12}$$

In Eq. 12, *passed*(s) is the number of passed test cases that executed statement *s* one or more times. Similarly, *failed*(*s*) is the number of failed test cases that executed statement *s* one or more times. *totalpassed* and *totalfailed* are the total numbers of test cases that pass and fail, respectively, in the entire test suite.

SOBER is a statistical model-based fault localization approach proposed by (Liu and Han (2006); Liu et al. (2005). Unlike existing statistical debugging approaches that select predicates correlated with program failures, SOBER models evaluation patterns of predicates in both correct and incorrect runs respectively and regards a predicate as bug-relevant if its evaluation pattern in incorrect runs differs significantly from that in correct ones.

CT (Cause Transitions) was proposed by Cleve and Zeller (2005). A cause transition is where a cause originates-that is, it points to program code that causes the transition and hence the failure. Thus, a cause transition is a candidate for a code correction-and cause transitions can be isolated automatically, just like causes in the program state.

The PPDG based fault localization approach (RankCP) (Baah et al., 2010) judges the fault probability of a node based on the conditional probability of this node given the state of their parent nodes. This is achieved by computing the conditional probability of a node's current state (x_{ji}) given the current state configuration (pa_{ji}) of its parents (i.e., $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$), which reflects how the parent nodes influence their children nodes.

5.4. Experiment results

Fig. 6 shows the experimental results on *Siemens* programs. We obtained the fault localization results for RankCP, Tarantula, SOBER and CT from published papers (Baah et al., 2010; Jones and Harrold, 2005; Liu and Han, 2006; Liu et al., 2005; Cleve and Zeller, 2005). It shows the percentage of faults that can be located when a certain percentage of code is examined. The horizontal axis represents the percentage of a program's statements that must be examined to find the faults and the vertical axis represents the percentage of faulty versions that are found given a score on the horizontal axis. The lower the percentage of statements examined, the higher the effectiveness of fault localization approach is. As shown in Fig. 6, the curves of BNPDG and RankCP lie above the curves of other approaches, indicating that BNPDG and RankCP are more effective. This is significant because finding out the abnormal dependence relationships in the execution of the program by probabilistic graphical model can help to locate the fault more effectively. When less than 1 percent of the code must be examined, our approach is approximately 1.7, 3.8, 6.3, and 11.4 times more effective than RankCP, Tarantula, SOBER and CT.

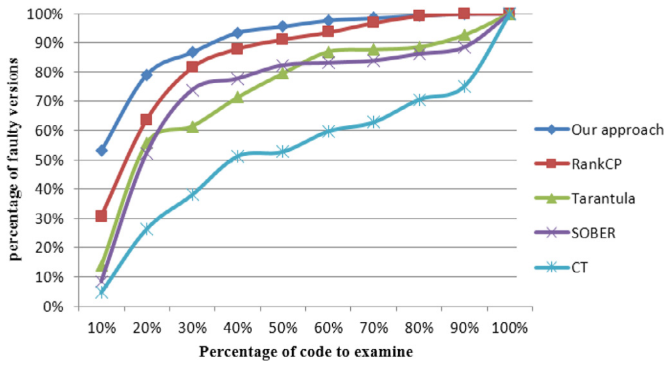


Fig. 6. Cumulative comparison with other approaches on *Siemens suite.*

Table 3 shows the detailed results of our approach on each subject *Siemens* program. We can observe that our approach is effective in locating the fault. In many programs, less than 30% statements must be examined to find all faulty versions.

Since both our approach and RankCP adopt the probabilistic graphical model and the PDG, we compare our approach and RankCP on the *Space* program for the scalability study. Table 4 demonstrates the detailed comparison results. The column "MBT" gives the time to build BNPDGs. The column "BNPDG size" gives the number of nodes in BNPDG in the faulty version. In order to give a detailed view of the effectiveness of our approach, the column "our approach" gives the number of nodes in the BNPDG that must be examined to find the faulty statement instead of the value of *Score*. As shown in Table 4, the number of statements examined until the faulty node is found using RankCP is higher than our approach in most cases. The column "MBT" shows that our approach requires less than 15 min to build the BNPDG, which indicates that our BNPDG-based fault localization approach can scale to larger programs. In additional, we compute the effect size, Cohen's *d* Hassan et al., 2013), to quantify the amount of difference between our approach and RankCP. For Cohen's *d*, we can obtain *d* from formula ((13) and (14):

$$d = \frac{\overline{X_1} - \overline{X_2}}{S_p} \tag{13}$$

$$S_p = \sqrt{\frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{(n_1 - 1) + (n_2 - 1)}} \tag{14}$$

where $\overline{X_1}$ is the sample mean of group 1, $\overline{X_2}$ is the sample mean of group 2, S_p is the pooled standard deviation, n_1 is the number of samples of group 1, n_2 is the number of samples of group 2, S_1^2 is the variance of group 1 and S_2^2 is the variance of group 2. Normally, the effect size is small if $0 < d < 0.2$, the effect size is medium if $0.2 < d < 0.8$, and the effect size is large if $d > 0.8$. The effect size of our approach-RankCP is 0.3582, which indicates that the performance of our approach has a medium effect than that of RankCP.

Table 3
Percentage of located faults to the percentage of code examined.

Score	Print_tokens	Print_tokens2	Replace	Schedule	Schedule2	tcas	tot-info
0–1%	42.85	60.00	62.50	44.44	50.00	43.90	60.86
1–10%	28.57	10.00	21.88	33.33	30.00	36.59	13.04
10–20%	14.29	10.00	6.25	22.22	10.00	4.88	4.35
20–30%	24.29	20.00	6.25	0.00	10.00	2.44	8.70
30–40%	0.00	0.00	3.13	0.00	0.00	4.88	4.35
40–50%	0.00	0.00	0.00	0.00	0.00	2.44	4.35
50–60%	0.00	0.00	0.00	0.00	0.00	2.44	0.00
60–70%	0.00	0.00	0.00	0.00	0.00	2.44	0.00
70–80%	0.00	0.00	0.00	0.00	0.00	0.00	4.35
80–90%	0.00	0.00	0.00	0.00	0.00	0.00	0.00
90–100%	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 4
Comparison resutls of the scalability case study on the space subject.

Faulty Version	MBT (seconds)	BNPDG size	Our approach	RankCP
V10	743	4211	182	193
V11	756	4211	1231	1458
V12	807	4211	1423	1573
V13	783	4211	9	11
V14	694	4210	1032	1071
V15	532	4210	1130	1146
V16	782	4211	53	61
V17	793	4210	126	130
V18	813	4211	1353	1554
V19	701	4213	1	1
V20	801	4211	1532	1664
V21	792	4211	1422	1635
V22	813	4206	1	1
V23	788	4211	15	20
V24	789	4196	83	101
V25	501	4209	1243	1263
V26	643	4211	832	945
V27	874	4211	1743	1808
V28	374	4211	1001	1017
V29	784	4211	174	271
V30	103	4211	39	42
V31	710	4211	21	24
V33	803	4211	1302	1632
V34	1	4211	15	10
V35	792	4203	1394	1370
V36	821	4211	309	433

6. Validity threats

In this section, we discuss several validity threats that may have an impact on the results of our studies.

External validity. Threats to external validity occur when the results of our experiments cannot be generalized. As a preliminary study, we conduct our experiments on the *Siemens suite* and *Space* to explore the generality of our approach. Although these datasets have been widely used in many fault localization studies, we still cannot claim that our approach can be generalized to other datasets. We will address this threat in future work. Nevertheless, this work provides a detailed algorithm description. Therefore, other researchers can easily replicate our approach on new datasets.

Internal validity. Threats to internal validity mainly come from the incorrect program implementation-specifically, the process of generating a BNPDG that may affect the experimental results. To overcome these threats, we have compared the manually generated BNPDGs of small subjects to their BNPDGs generated automatically by our techniques to ensure they match each other.

Construct validity. Threats to construct validity concern the appropriateness of the metrics used in our experiments. We used the *score* metric to measure the effectiveness of the proposed approach because it is the metric used by many other fault localization approaches. However, it is difficult to determine whether it conforms to the way in which programmers locate the faults in the program. Therefore, more studies are required to determine the appropriateness of the metric for measuring the effectiveness of fault localization approaches.

Conclusion validity. Threats to conclusion validity focus on the statistical analysis method. In this work, we use the effect size, Cohen's *d* to statistically quantify the amount of difference between two approaches.

7. Related work

In this section, we briefly review the existing fault localization approaches. These approaches can be categorized into three main types: program slicing, statistical analysis and probabilistic graphical model.

7.1. Program slicing based fault localization

Program slicing includes static slicing (Lyle and Weiser, 1987; Zhang and Santelices, 2016), dynamic slicing (Zhang and Gupta, 2007; Hofer and Wotawa, 2012; Wen et al., 2011; Alves et al., 2011) and execution slicing (Wong and Qi, 2006). The static slicing of an incorrect variable at a program execution point includes all those program statements which possibly influence the value of the variable at that point. In contrast, the dynamic slicing of an incorrect variable at a program point is the set of executed statements which actually affect the value of the variable at the given program point under some execution. The execution slicing is the set of code executed by a given test cases. By studying the program slicing of the incorrect value, a developer can eliminate the irrelevant value and narrowing search area to detect the faulty statements. However, there may still be too much code that needs to be examined. In addition, the slicing-based fault localization approach does not provide a ranking of the statements in the slices presented to the developer.

7.2. Statistical analysis based fault localization

This statistical analysis-based fault localization approach locates fault-relevant statements by comparing the statistical differences of program elements in passed and failed test cases. Tarantula (Jones and Harrold, 2005) adopts the pass/fail statuses of test cases and events occurred during execution of each test case to offer the developer recommendations of what may be the faults that are causing test-case failures. SOBER (Liu and Han, 2006; Liu et al., 2005) is a statistical model-based approach for localizing software bugs without any prior knowledge of program semantics. The approach in (Gong et al., 2015) utilizes program slices of a set of test runs to capture the influence of a program entity's execution on the output, and uses statistical analysis to measure the suspiciousness of each program entity being faulty. A state dependency probabilistic model in (Abdollahi et al., 2016) for fault localization was proposed. The approach can capture the behavior state information during program execution, and the fault-localization approach differentiates the state dependencies in passed and failed test cases.

7.3. Probabilistic graphical model based fault localization

The probabilistic graphical model based fault localization approach adopts the probabilistic graphical model and the PDG. Their difference lies in the choice of the probabilistic graphical model, e.g. the BN, the Markov network, the dependency network and the causal graph. The PPDG (Baah et al., 2010) based on the dependency network facilitates probabilistic analysis and reasoning about uncertain program behavior associated with the fault. Another work (Baah et al., 2010) using the causal graph for fault localization obtains causal-effect estimates that are not subject to confounding bias. The divergences between different methods based on probabilistic graphical models depend on aspects of whether issues are suitable for them, their theoretical foundations, and their inference abilities. The BN for the BNPDG has comparative advantages over these aspects.

8. Conclusion and future work

In this paper, we proposed a novel probabilistic graphical model called Bayesian Network based Program Dependence Graph (BNPDG) that takes the output node as the common condition to calculate the conditional probability of each non-output node. The ability of the BNPDG to reason the probability of suspicious nodes across nonadjacent node makes it more effectively than PPDG in fault localization. We conducted experiments on the *Siemens suite* and *Space* datasets to evaluate the performance of the proposed approach. The experimental results indicate that our approach has stronger inference capability on program dependences, which leads to more accurate and scalable fault localization.

In the future, we will validate the generalization of our approach on more real-world datasets. We also plan to apply BNPDG to other software engineering tasks.

Acknowledgment

This work is partly supported by the grants of National Natural Science Foundation of China (61572374, 61070013, 61300042, U1135005, 61303025, 61440054).

References

Abdollahi, A., Pattipati, K.R., Kodali, A., Singh, S., Zhang, S., Peter, B.Luh, 2016. Probabilistic Graphical Models for Fault Diagnosis in Complex Systems. In: Principles of Performance and Reliability Modeling and Evaluation, pp. 109–139.

Alves, E., Gligoric, M., Jagannath, V., Amorim, M., 2011. Fault-localization using dynamic slicing and change impact analysis. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 520–523.

Baah, G.K., Podgurski, A., Harrold, M.J., 2010a. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. Softw. Eng.. IEEE Comput. Soc. 36 (4), 528–545.

Baah, G.K., Podgurski, A., Harrold, M.J., 2010b. Causal inference for statistical fault localization. In: Proceedings of 19th international symposium on Software testing and analysis, pp. 73–84.

Chen, L, Ma, W, Zhou, Y, et al., 2016. Empirical analysis of network measures for predicting high severity software faults. Sci. China Inf. Sci. 59 (12), 122901.

Chen, T.Y., Xie, X., Kuo, F.C., et al., 2015. A revisit of a theoretical analysis on spectrum-based fault localization. In: Proceedings of Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual. IEEE, 1, pp. 17–22.

Cleve, H., Zeller, A., 2005. Locating causes of program failures. In: Proceedings of the 27th international conference on Software engineering, pp. 342–351.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. In: Proceedings of ACM Transactions on Programming Languages and Systems, 9. ACM, pp. 319–349.

Gong, D., Su, X., Wang, T., Ma, P., Yu, W., 2015. State dependency probabilistic model for fault localization. Inf. Softw. Technol 430–445.

Hammer, C., Snelting, G., 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. 8 (6), 399–422.

Hassan, AE, Hindle, A, Runeson, P, et al., 2013. Roundtable: what's next in software analytics. IEEE Softw. 30 (4), 53–56.

Hofer, B., Wotawa, F., 2012. Spectrum enhanced dynamic slicing for better fault localization. ECAI 420–425.

Hristova, M., Misra, A., Rutter, M., et al., 2003. Identifying and correcting Java programming errors for introductory computer science students. ACM SIGCSE Bullet. 35 (1), 153–156.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 273–282.

Laski, J.W., Korel, B., 1983. A data flow oriented program testing strategy. IEEE Trans. Softw. Eng. (3) 347–354.

Liu, C., Han, J., 2006. Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 46–56.

Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S., 2005. Sober: Statistical Model-Based Bug Localization. In: In joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 286–295.

Lyle, J.R., Weiser, M., 1987. Automatic program bug location by program slicing. In: Proceedings of the 2nd International Conference on Computer and Applications, pp. 877–883.

Neapolitan, R.E., 2003. Learning Bayesian Networks. Prentice Hall.

Peng, H., Ding, C., 2003. Structure search and stability enhancement of Bayesian networks. In: Proceedings of 3rd IEEE International Conference on Data Mining, pp. 621–624.

Reshef, D.N., Reshef, Y.A., Finucane, HK, et al., 2011. Detecting novel associations in large data sets. Science 334 (6062).

Wen, W.Z., Li, B.X., Sun, X.B., Li, J.K., 2011. Program slicing spectrum-based software fault localization. In: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, pp. 213–218.

Wong, W.E., Qi, Y., 2006. Effective program debugging based on execution slices and inter-block data dependency. J. Syst. Softw. 79, 891–903.

Xie, X., Chen, T.Y., Kuo, F.C., et al., 2013b. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. 22 (4), 31.

Xie, X., Kuo, F.C., Chen, T.Y., et al., 2013a. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In: Proceedings of ACM International Symposium on Search Based Software Engineering. Berlin Heidelberg. Springer, pp. 224–238.

Xie, X., Wong, W.E., Chen, T.Y., et al., 2013c. Metamorphic slice: an application in spectrum-based fault localization. Inf. Softw. Technol. 55 (5), 866–879.

Zhang, X., Gupta, N., 2007. Locating faulty code by multiple points slicing. Softw.: Pract. Exp. 37, 935–961.

Zhang, Y., Santelices, R., 2016. Prioritized static slicing and its application to fault localization. J. Syst. Softw. 114, 38–53.

Zhou, YM, Leung, H, Song, QB, et al., 2012. An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. Sci. China Inf. Sci. 55 (12), 2800–2815.

Xiao Yu is currently pursuing the doctoral degree with the Wuhan University, Wuhan, China. His current research interests include software engineering and data mining.

Jin Liu received his Ph.D. degree in Computer Science from Wuhan University in 2005. He is a professor in State Key Laboratory of Software Engineering, Compute School, Wuhan University, China. His research interests include software analytics and software repository mining.

Zijiang Yang received his Ph.D. degree from the University of Pennsylvania. He is a professor of Computer Science at Western Michigan University. His research is in the broad areas of software engineering and formal methods. The primary focus is to develop formal method based tools to support the debugging, analysis and verification of complex systems.

Xiao Liu received his Ph.D. degree in Computer Science and Software Engineering from the Faculty of Information and Communication Technologies at Swinburne University of Technology in 2011. He is currently a Senior Lecturer at School of Information Technology, Deakin University, Melbourne, Victoria, Australia. His research interests include workflow systems, cloud computing and social network.