A Prioritization Approach for Software Test Cases Based on Bayesian Networks

Siavash Mirarab and Ladan Tahvildari

Department of Electrical and Computer Engineering,
University of Waterloo, Ontario, Canada N2L 3G1
{smirarab,ltahvild}@uwaterloo.ca

Abstract. An important aspect of regression testing is to prioritize the test cases which need to be ordered to execute based on specific criteria. This research work presents a novel approach to prioritizing test cases in order to enhance the rate of fault detection. Our approach is based on probability theory and utilizes Bayesian Networks (BN) to incorporate source code changes, software fault-proneness, and test coverage data into a unified model. As a proof of concept, the proposed approach is applied to eight consecutive versions of a large-size software system. The obtained results indicate a significant increase in the rate of fault detection when a reasonable number of faults are available.

Keywords: Test case Prioritization, Regression Testing, Bayesian Networks.

1 Introduction

Prioritizing existing test cases from earlier versions of software is one of the main techniques used to address the problem of regression testing. Regression testing is considered as one of the most expensive tasks in software maintenance activities [1]. Such a technique uses the test-suite developed for an earlier version of a software system to conform the new added requirement in the current version. Selecting all or a portion of the test-suite to execute which is referred to as Regression Selection Techniques (RST) can be very costly [2,3,4]. Furthermore using RST, testers do not have the option to adjust their test-effort to their budget. To provide the missing flexibility, researchers have introduced prioritization techniques [5,6] by means of which testers can order the test cases based on certain criteria, and then run them in the specified order and as much as they can afford. To further assist testers to adjust the cost and effort, models of cost-benefit analysis are introduced [7]. During the past ten years, there has been much research on techniques of prioritization [8,9,10,11,12,13].

Despite all the above-mentioned research activities, empirical studies indicate that there is a significant gap between optimal solutions to prioritization problem and proposed techniques [9]. Also, they show that the performance of different techniques depend largely on the target software system. Furthermore, one can imply that techniques using more than one factor typically perform better than those with one criterion. Therefore, to fill the mentioned gap, we need to build techniques which can incorporate a diverse range of available data, from test coverage to history of fault detection.

In this paper, we present a novel test-suite prioritization framework which integrates various sources of information into one single model. Our technique is based on a probabilistic specification of the problem. Similar to the definition of Kim *at el.* [10], our prioritization approach is based on ordering test cases according to their success probability. The proposed process uses conditional probability and utilizes a Bayesian Network [14] model which takes advantage of source code modification information, univariate measures of fault-proneness, and test coverage data. We evaluate the performance of our technique using APFD (Average Percentage Faults Detected) [5] measure on eight consecutive versions of a large-size Java application augmented with hand-seeded faults; we then compare our results to some of the common techniques from literature. The results show that when there are reasonable number of faults in the source code, our proposed novel technique is capable of achieving better values of APFD in the comparison with other techniques.

The rest of the article is organized as follows: Section 2 deals with the problem statement. Then, our proposed approach to solve the problem is presented. Section 4 gives a brief introduction to Bayesian Networks (BN) while elaborating on how we have designed our BN model. Section 5 discusses the obtained results after applying our techniques on a large size case study. Finally we make conclusion and point out some future directions for this research work.

2 Problem Statement

The classic definition of test case prioritization is based on finding a permutation of test cases which can maximize an award function [5]. Kim *et al.* look at the same problem from a probabilistic point of view. They describe prioritization as: *i*) applying an RST technique, *ii*) assigning a selection probability to each of the remaining tests, *iii*) drawing a test case using the assigned probabilities, and *iv*) repeating until time is exhausted. Adopting the probabilistic nature of their description and with some modification, a prioritization can be considered as:

1. Gathering all "useful" evidences $\mathcal{E}_i$ from software system.
2. Using a "prioritization technique" to assign a probability of success to each test case t_i in test-suite $\mathbb{T}$, given all the evidences $P(\mathcal{T}_j|\mathcal{E}_1,\ldots,\mathcal{E}_n)$.
3. Selecting and running a test case from $\mathbb{T}$ based on the defined probability model.
4. Updating the $P(\mathcal{T}_j|\mathcal{E}_1,\ldots,\mathcal{E}_n)$ values when applicable.
5. Repeating step 4 until release criteria is met.

We do not think of the first step in Kim's description as a part of prioritization; thus, it is taken out. Moreover, in this approach some sort of selection is automatically done when low probabilities are assigned to test cases. However, in practice one may prefer to apply some less costly selection techniques before prioritization in order to reduce the size of test-suite. "Useful" evidence in step 1 means all information the "prioritization technique" of step 2 is interested in (e.g. test coverage information, code change). In step 2, the main part of this process, we have a set of random variables $\mathbf{t_i}$, each of which reflect the outcome of a test case t_i from $\mathbb{T}$. These variables have two possible values of "Success" (meaning that a defect is detected) and "Failure". The event of "Success" in t_i is denoted as $\mathcal{T}_i$. A "prioritization techniques" is a systematic way to estimate all $P(\mathcal{T}_j|\mathcal{E}_1, \ldots, \mathcal{E}_n)$s. Step 3 is also slightly different than Kim's. Although the notion of drawing test cases using assigned probabilities is interesting in that it gives all test cases some selection chance and helps discover the residual faults, it is not the only option. One may simply order test cases according to their probability, particularly when experimentation is involved deterministic nature of the later approach is more appropriate. Step 4 is an important addition to previous definitions which provides a feedback mechanism to add the learned information from each test execution (e.g. its outcome) to the model defined in step 2. Note that this mechanism includes, but is not limited to the techniques of [5] where feedback is used while ordering test cases not after each run. Finally, in step 5 we generalize the stopping condition to "met release criteria" which can be anything from testing time exhaustion to reliability requirement satisfaction [15].

The proposed view of prioritization can be applied to existing techniques. In the following, two families of existing techniques are briefly described in accordance with the aforementioned view of prioritization problem.

- **Coverage-based Techniques.** The most important aspect of any technique is the set of evidences it takes advantage of. In this family of prioritization techniques, the evidence variable is the number of code elements that are covered by each test case. For example, at method level coverage-based technique, the number of covered methods is used to estimate the probability of success for each test case:

$$P(\mathcal{T}_j|\mathcal{E}) = \frac{the\ number\ of\ methods\ covered\ by\ \mathrm{t}_i}{total\ number\ of\ methods}$$

- **Change-coverage Techniques.** In this family of techniques both information of test coverage and source code change are used as the evidence. At block level, for example, the estimation of success probability is:

$$P(\mathcal{T}_j|\mathcal{E}) = \frac{the\ number\ of\ changed\ blocks\ covered\ by\ \mathrm{t}_i}{total\ number\ of\ changed\ blocks}$$

The main part of the described process is step 2 where one should estimate the probability of success for each test case. The problem that this paper addresses is to build such techniques. In the following section, our approach to solve this problem is described.

3 Proposed Approach

Our approach addresses the prioritization problem by: *i*) extracting different sets of evidence from the source code, *ii*) integrating all information to a single Bayesian network model, and *iii*) using probabilistic inference to compute $P(\mathcal{T}_j|\mathcal{E})$ values. Fig. 1 illustrates a high-level schema of this approach. The first

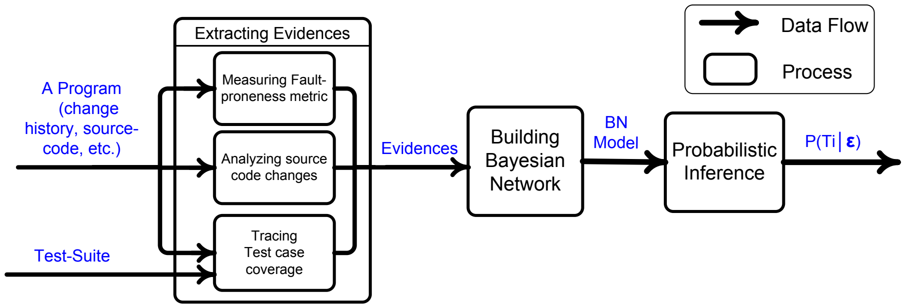

Extracting Evidences
Data Flow
Process
A Program (change history, source-code, etc.)
Measuring Fault-proneness metric
Analyzing source code changes
Tracing Test case coverage
Test-Suite
Evidences
Building Bayesian Network
BN Model
Probabilistic Inference
P(Ti | ε)

Fig. 1. Three Phases of Proposed Approach

step in performing prioritization is to gather all useful information that is to be included in the model. Our current solution exploits three sources of information: *software quality metrics*, *test coverage measures*, and *change analysis data*. *Extracting evidences* is undertaken in order to provide the necessary data for the next phase, *Building Bayesian Network*, in which an inclusive probabilistic model to relate these data is built. The details and rational behind using these evidences will be covered in the next section, where we give an in-depth description of the second phase. The last phase is to employ the probabilistic inference algorithms to associate to each test case its probability of success given the collected evidences. Note that the first and last phases are well-established research works and here we just make use of the existing contributions to implement them.

4 Building Bayesian Network

In this section, first we introduce Bayesian Networks briefly and then elaborate further on the second phase of our proposed process through which the other phases are more clarified.

4.1 Background: Bayesian Network

Bayesian Network (BN) is a special type of "probabilistic graphical models" [14]. A BN is a directed acyclic graph consisting of three elements: *nodes* representing random variables, *arcs* representing probabilistic dependency among those variables, and *Conditional Probability Distribution Table (CPT)* for each variable, given its parents. The nodes can be either evidence or latent variables.

An evidence variable is a variable of which we know its values (i.e. it is measured). Arcs specify the causal relation between variables. Each node has a table which includes the probabilities of outcomes of its variable given the values of its parents.

Bayesian networks reflect the belief of experts about the problem domain. They can be used to answer probabilistic queries. For example, based on the evidence (observed) variables, the posterior probability distributions of some other variables can be computed (probabilistic inference). However, designing a BN model is not a trivial task. There are two facets to modeling a BN, designing the structure and computing the parameters. Regarding the first issue, the notions of conditional independence and causal relation [16] can be of great help. Intuitively, two events (variables) are conditionally independent if knowing the value of some other variables makes the outcomes of those events independent. It is important to make sure that conditionally independent variables are not connected to each other. Designing based on causal relationships is one way to achieve that. For computing the parameters, expert knowledge, statistical learning, and probabilistic estimations can be used. One potential problem is that we may know how a variable is dependent on each of its parents, but do not have its distribution conditioned on all parents. In these situations, "noisy-OR" assumption can be helpful. The noisy-OR assumption gives the interaction between the parents and the child a causal interpretation and assumes that all causes (parents) are independent of each other in terms of their influence on the child [14]. More formally, this assumption asserts whatever prevents one parent to cause a child is independent from what prevents the other parents to cause the child.

4.2 Proposed BN Model

Empirical studies conducted in the literature indicate that an important factor in performance of a technique is the evidences it utilizes [9]. The rational behind using Bayesian networks for prioritization is to unify various types of evidences in one single model.

As mentioned in Section 4.1, modeling is the main focus in solving the problems using BN. A description of how three basic elements of a BN is designed in our approach follows:

Nodes. There are three categories of nodes in these models:

- **ce:** These variables represent change in the elements of the program. Each software element in the considered level of granularity (i.e. a class) has a node of this type. These variables can take a value of "Changed" or "Unchanged".
- **fe:** This category reflects our belief whether each element is faulty. Similar to the previous family, each element of the program has one node and each node can have the values of "Faulty", or "Non-Faulty".
- **t:** These variables represent the outcome of a test case which can be "Success" or "Failure". Each test case has one node of this type and the probability distributions of these nodes are what we are looking for $P(\mathcal{T}_i|\mathcal{E})$.

Arcs. Each arc in a BN indicates a causal relation between variables of two connected nodes. There are two set of arcs in our network:

- **ce** – **fe:** Each **fe** node is the child of the corresponding (i.e. of the same code element) **ce** node. The existence of these arcs reflect the causal relation that changes to elements of software can introduce faults in the same element.
- **fe** – **t:** Each **t** node is the child of some **fe** nodes. These arcs imply the causal relation between presence of fault in a software element and success of test cases that examine that element.

In Fig. 2 the overall structure of the designed model is illustrated. Each **ce** node is connected to one **fe** node and the **fe** nodes are connected to an arbitrary number of **t** nodes.

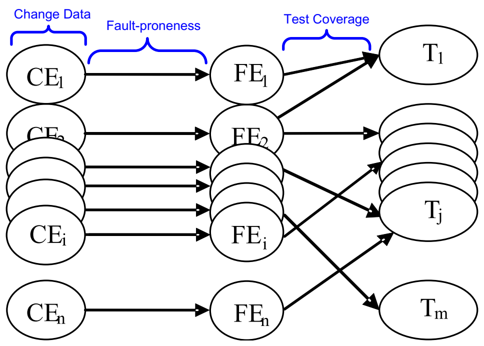


Fig. 2. The Structure of the Bayesian Network

CPT. Each category of nodes has its own Conditional Probability Table:

- $P(\mathbf{ce}_i)$**:** **ce** nodes are not the child of any other node, so their distribution is not conditional. These nodes are the only "observed" variables of the model. In a simple model, $P(\mathbf{ce}_i = Changed)$ can be set to either 0 or 1, meaning that software is either "Changed" or "Unchanged". However, they also can be used to represent the amount of change an element has gone through. In this case $\mathbf{ce}_i$ variables mean the effective change of the element and $P(\mathbf{ce}_i = Changed)$ reflects our belief that the element has been effectively changed.

$$P(\mathbf{ce}_i = Changed) = ChangeIntensity(e_i)$$

In this formula, $ChangeIntensity(e_i)$ is a function which returns how much semantic change the element e_i has gone through. This function can be implemented with algorithms as simple as Unix diff. In our study, we have used an algorithm presented in [17] which uses byte code to estimate similarity between two versions of a program.

• $P(\mathbf{fe}_i|\mathbf{ce}_i)$ **:** Each **fe** nodes is a child of one and only one parent, which is the corresponding **ce** variable. Considering that both **fe** and **ce** can take two values, the CPT will contain 4 values, two of which are trivial, since $P(\mathbf{fe} = \mathit{Faulty}|\mathbf{ce}) = 1 - P(\mathbf{fe} = \mathit{Non\text{-}Faulty}|\mathbf{ce})$. Therefore, we need to estimate two values: $P(\mathbf{fe} = \mathit{Faulty}|\mathbf{ce} = \mathit{Changed})$ and $P(\mathbf{fe} = \mathit{Faulty}|\mathbf{ce} = \mathit{Unchanged})$. In general, the probability of presence of fault in software is called fault-proneness and is profoundly studied in literature [18,19]. It is empirically shown that one can approximately predict the fault-proneness of code elements using software metrics. To build these models of fault-prediction, there are two major options: *multivariate* and *univariate* models. Univariate models estimate fault-proneness using one single metric. Multivariate models, on the other hand, are a linear combination of univariate models. To use multivariate models, one should "train" the model on a second program and apply the potentially biased model to the system in question. As empirically evaluated in [18], using this approach, multivariate models do not necessarily generate better results; thus, in this work we use univariate models. The aforementioned studies (and also an empirical study on the relation between APFD and software metrics [20]) indicate that measures of complexity and coupling are better indicators of fault-proneness. One specific study [19] has shown that coupling is a significantly better measure than other metrics. Although our model can fit in any software metric, here we use measures of coupling as an indicator fault-proneness:

$$P(\mathbf{fe}_i = \mathit{Present}|\mathbf{ce}_i = \mathit{Changed}) = \frac{\alpha\, CBO(e_i)}{\max(CBO(e_x))} + \delta_1,\ (\alpha + \delta_1 \leq 1)$$

In this formula, e_i is an element of the system, say a class, and CBO (Coupling between Objects) is an object-oriented metric from Chidamber and Kemerer suite [21] which counts the number of classes to which a given class is coupled (i.e. uses its methods and/or fields). The choice of this metric is based on the mentioned empirical studies. The dominator is a normalization factor and α and δ_1 bound the probability of fault introduction.

As for $P(\mathbf{fe}_i = \mathit{Faulty}|\mathbf{ce}_i = \mathit{Unchanged})$, estimating this value is tricky because it represents the less probable situation that an element is faulty, even though it is not changed. This can happen because of residual faults (from previous versions) or because of the impact of changes in other elements. Estimating both causes is hard and calls for more thorough empirical studies. In current modelling we use the following formula:

$$P(\mathbf{fe}_i{=}\mathit{Faulty}|\mathbf{ce}_i{=}\mathit{Unchanged}) = \frac{\beta\, f\text{-}out(e_i)}{\max(f\text{-}out(e_x))} + \delta_2\,,\ (\beta + \delta_2 \ll \alpha + \delta_1 \leq 1)$$

Here, *f-out* (fan out), is a measure of the number of classes an specific class is coupled to (i.e. uses them) [22]. Using this metric is mostly in order to capture change impacts. As known, the more fan out a class has, the more it is endangered by changes of other classes. The important invariant is that the probability of fault presence in unchanged elements should be much less than in changed elements. Let $\gamma = \frac{\alpha+\delta 1}{\beta+\delta 2}$. By adjusting γ (the change effect factor) we can control the degree to which the presence of change in an element raises our belief in its fault-proneness.

- $P(\mathbf{t}_i|\mathbf{fe}_1 \ldots \mathbf{fe}_n)$ **:** Unlike the other node types, **t** nodes can have more than one parent. That is due to the fact that a test case may be able to find faults from different elements of the software. These values can be determined according to the coverage information of a test case. Normally, the information of test coverage is available only when test cases are executed and prioritization does not have any justification. The solution is to use the coverage information from previous versions for the current program. To further enhance the reliability of this solution, one may use heuristics of [12] (however, we believe even without these heuristics, the estimation is reasonable). Having test coverages, we estimate:

$$P(\mathbf{t}_i = \mathit{Success}|\mathbf{fe}_j = \mathit{Faulty}) = \mathit{Cov}(t_i, e_j)$$

Where $\mathit{Cov}(t_i, e_j)$ is a function returning the percentage of the code element j covered by test case i. This formula estimates the relation between a test case and one single element. However, to build the CPT we need the probability of success for a test given all combinations of values of **fe** variables. In this situation, the table would become enormous and its size would grow exponentially with the number of covered elements by a test. In order to cope with this problem, we make the noisy-OR assumption, explained in Section 4.1. The assumption is that the relation of a test case to an element is independent from its relation to any other element. It can be argued that the ability of a test case to reveal a fault in one element is not related to its fault revealing ability in other elements, hence the assumption. Having the noisy-OR assumption we can say:

$$P(\mathbf{t}_i|\mathbf{fe}_1 \ldots \mathbf{fe}_n) = 1 - (1 - P_0) \prod_j \frac{(1 - P(\mathcal{T}_i|\mathbf{fe}_j = \mathit{Faulty}))}{1 - P_0} \tag{1}$$

In this formula, P_0 is $P(\mathcal{T}_i|\mathbf{fe}_1 = \mathit{Non\text{-}Faulty}, \ldots, \mathbf{fe}_n = \mathit{Non\text{-}Faulty})$. This value (also called "leak") is the probability of a test case succeeding even though all of its related elements are non-faulty. This value should be zero unless the information of coverage is incomplete or there exists an integration fault. For consideration of these causes, a very small constant can be assigned to leak parameter. Formula 1 is a straight-forward usage of noisy-or formula (justification of this formula can be found in BN references such as [16]).

In this model **ce** nodes are observations and to estimate the distribution of **t** nodes (the desired variables), we need to perform probabilistic inference. There are many inference algorithms introduced for BNs which generally fall into two categories of exact and sampling algorithms. Due to well-structured nature (three layers of independent variables) of our network, we can use exact inference even for large systems. Details of how inference algorithms work are out of the scope of this paper but the general idea is first compiling the directed graph into a tree, and then updating the probabilities in the tree.

5 Experiment

To evaluate the proposed approach, we have built a semi-automated environment for test case prioritization. As a proof of concept, eight consecutive versions of Apache Ant [23] with a catalogue of 10 prioritization techniques are examined.

5.1 Prioritization Environment

To assist future experiments of test case prioritization, a semi-automated framework is implemented. Fig. 3 depicts a high-level schema of this system.

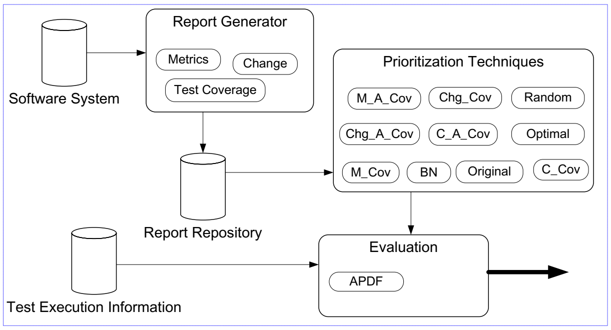


Fig. 3. The High-Level Schema of the Framework

The first subsystem, “Report Generator”, is mostly implemented using external tools. To collect software metrics, a tool called ckjm [24] is used; for gathering coverage information Emma [25] is utilized and the change information is obtained from Sandmark [17] which is a watermarking program but provides interesting change track algorithms. In the second subsystem,“Prioritization Techniques”, different techniques of prioritization are implemented. This subsystem uses generated reports from the first subsystem. For implementing the BN technique we have used Smile Library [26]. Finally, the last subsystem, “Evaluation”, is responsible for measuring evaluation metrics (merely APFD).

5.2 Experiment Setup

Subject Program. Performing experiments in test case prioritization calls for many artifacts some of which are very expensive to gather. In particular, the subject program needs to have many faulty versions. Do *et al.* [27] have built a repository of C and Java open source programs with hand seeded faults called “Software-artifact Infrastructure Repository (SIR)”. From their Java repository,

Table 1. Eight Consecutive Versions of Apache Ant

Metric Name	v0	v1	v2	v3	v4	v5	v6	v7
Faults Count	0	1	1	2	4	4	1	6
Test Case Count	0	28	34	52	52	101	104	105
Number of Classes	143	229	343	343	533	537	537	650
Lines Of Code (K)	23	37	57	57	95	97	97	124

Apache Ant has the most number of seeded faults and has a reasonable size (Table 1). We have used this program for all of our experimentation.

Evaluation Metric. To be able to compare our results to other empirical studies (esp. those of [8] as one of the rare studies focusing on Java programs), APFD is used as the evaluation metric. This metric aims to calculate fault detection rate by measuring the weighted average of the percentage of faults detected over the test suite execution period. APFD values range from 0 to 100 and higher numbers indicate faster fault detection rates. Its precise definition can be found in [5]. However, this metric has some drawbacks, for example, it neither takes into account the cost of each individual test nor the severity of faults.

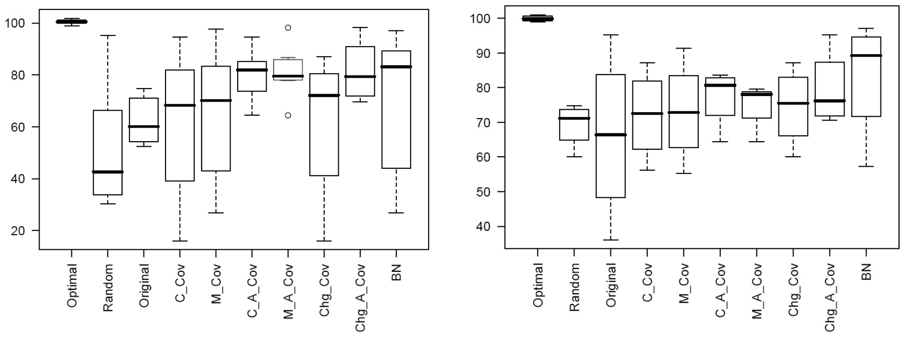
Prioritization Techniques. In this study, ten prioritization techniques are examined. Table 2 lists these techniques. The first three are control techniques: Optimal is the best possible order computed in a greedy manner; Random orders randomly (the average of 50 runs); and Original is the original order of test cases. The next six techniques are based on [8] and use coverage information. Their difference is in evidences used, granularity level, and use of feedback mechanism. Here, feedback means adjusting the coverage information after adding any test case to the order such that elements that are already covered do not affect next selections ([8]). Finally, BN represents our approach where the parameters are set as: $\alpha = 0.8$, $\delta_1 = \delta_2 = 0.1$, and $\gamma = 8$.

Table 2. Prioritization Techniques Used in the Experimentation

Name	Evidences	Level	Feedback
Optimal	Fault Matrix	N/A	Yes
Random	Nothing	N/A	No
Original	Nothing	N/A	No
C_Cov (Class Coverage)	Coverage	Class	No
M_Cov (Method Coverage)	Coverage	Method	No
C_A_Cov (Class Additional Coverage)	Coverage	Class	Yes
M_A_Cov (Method Additional Coverage)	Coverage	Method	Yes
Chg_Cov (Change Coverage)	Coverage+Change	Class	No
Chg_A_Cov (Change Additional Coverage)	Coverage+Change	Class	Yes
BN	All	Class	No

5.3 Discussion on Obtained Results

The results of the case study are depicted in Fig. 4. Almost all techniques perform better than "random" and "original" (the two control techniques). As far as the level of granularity for coverage information is concerned, there is no meaningful difference between class level and method level techniques. This result is in accordance with past empirical studies (although to our knowledge, class-level coverage were not previously inspected), and suggests using class level coverage information which is much easier to obtain. Also, it is evident that techniques employing the feedback mechanism (or "additional techniques") bring about better results. Although using change data leads to a 2% increase in the average APFD value (Table 3), there is no strong evidence that they outperform techniques with merely coverage information.



(a) All Versions (b) Versions with More Than One Fault

Fig. 4. Boxplot Diagram of the Experiment Results

As for BN technique, the median of its AFPD values among all versions is better than all other techniques (however not significantly). When considering average instead of median, although it is performing better than most of the techniques, it is not the best when we consider all the versions of the subject program (Table 3). More specifically, in average BN is performing better than all techniques without feedback mechanism, but worse than additional techniques. Note that BN as implemented in this experimentation does not take the advantage of any feedback. Taking a closer look at the data, we noticed that in many versions, BN technique is achieving the best performance. To inspect why the average performance of BN is not the best, the fault information of the subject program should be considered. There are three versions of the ant case study which are seeded only with one fault. The results indicate that on these three versions, BN achieves less fault detection rate than the other techniques. However, one can argue that one single fault does not provide a reliable basis for comparison of techniques. Thus, we took out the versions with one fault (three versions were such) and compared the results again.

Fig. 4 (b) illustrates that BN is performing better than all the other techniques, in this scenario. There is strong evidence that BN median values are better than all other techniques. Moreover, it results in better average APFD values. In average, it produces 5% better APFD values than the second best technique(*Chg_A_Cov*), 11% better than the average of all techniques except optimal, and 17% better than the original order (Table 3).

Table 3. Average and Standard Error of Different Techniques

Technique	**All Versions**		**More Than One Fault**	
Name	Average	SD	Average	SD
Optimal	100.47917	1.03495	99.90429	0.89719
Original	52.62632	24.43825	66.03252	24.57651
Random	62.61275	9.54245	69.28956	6.47141
C_Cov	60.18787	30.75548	77.35928	8.79435
M_COV	63.94816	27.70506	73.09323	14.88074
C_A_Cov	79.80473	10.55892	72.11833	13.09055
M_A_COV	81.44635	10.32284	75.01233	7.09485
Chg_Cov	59.79584	29.16008	74.55728	11.42936
Chg_A_Cov	81.84868	11.73580	79.56256	11.04775
BN	67.66124	29.40874	83.19689	17.88581

To further inspect the effect of the number of faults, we depicted APFD of the techniques versus the number of faults. Fig. 5 shows when the fault count of the system grows, the APFD value of "additional" techniques decrease; whereas the BN see an increase in the value of APFD. This suggests that feedback employing techniques perform better when a very small number of faults are available, but as the potential number of faults grows BN is the most promising technique. This result sounds rational because BN is a model based on probabilities and the more number of trials, the more reliable the results of a probabilistic model.

Fig. 5 also shows another interesting phenomenon. While all "additional" techniques have a negative slop, the techniques with no feedback mechanism all see an increase in APFD with number of faults. This observation should be empirically evaluated because when generalized, it has a very important practical implication: when the software is believed to contain many faults, the use of feedback is not useful but in more reliable systems, when testers struggle to find the last faults, feedback can improve the rate of fault detection.

In conclusion, BN technique seems to perform better than any other technique inspected here, when there are more number of faults. In three out of four versions with more than one fault, the BN produced the best results. Therefore, authors believe that the BN technique will perform very well when applied in practice to software systems that typically contain much more faults.

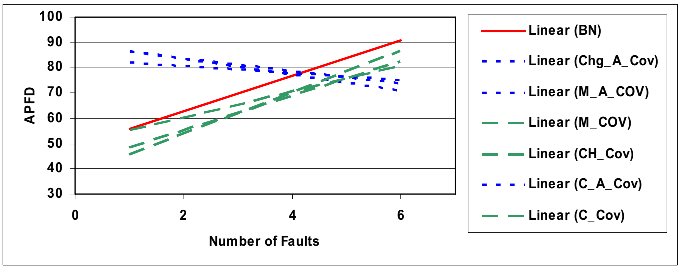


Fig. 5. APFD versus Fault Counts

6 Related Work

Many techniques for prioritization along with measures of assessing their performance have been introduced in literature. In [5], authors introduce APFD as a measure of fault detection rate and empirically evaluate their catalogue of techniques. In [9], more techniques with more than one criterion, are evaluated on larger case studies. In [8], the authors take a similar approach and evaluate similar techniques on Java programs and JUnit test cases.

Kim *et al.* [10] formulate test case prioritization based on the probability theory and focus on history-based prioritization to address the issue of continues software evolution and regression testing. They also introduce “total effort” and “fault-age” to measure cost-benefit trade-offs. Srivastava *et al.* have built Echelon [12] system to deal with prioritization in industrial environment. They propose extracting coverage information from byte-code for better performance and also provide some heuristics to address the high cost of gathering coverage information. Saff *et al.* [11] take a completely different approach by introducing continues testing. They developed a plug-in for Eclipse IDE and used developer behavior modelling to test software on the fly and during development. PORT [28] is another attempt in which potential defect severity and also issues related to testing of new code in regression testing are taken to account. More recently, Walcott *et al.* utilize Genetic Algorithms to solve the prioritization problem in a time-constrained situation [13].

On the other hand, employing Bayesian networks for testing has been addressed by some researchers. As early as 1997, in [29] authors described ways of modelling uncertainty in software into BN models that can be later used by testers and managers for either confirming, evaluating or predicting software uncertainties. In [30], authors use graphical models to provide a prediction model for the whole problem of software testing, and in [31] Bayesian networks is used to asses the overall software quality.

7 Conclusion and The Future Work

In this paper we first described test case prioritization problem from a probabilistic point of view, and then proposed a new approach to solve this problem using Bayesian networks. We introduced our framework to implement the approach and presented the results of a case study. The results suggest that the new approach can achieve high values of APFD, especially when the number of available faults are reasonable.

In the pursual of future research, first the results should be further inspected using empirical experiments and taking into account cost-benefit models. Also, the software faults in this case study are all hand-seeded and their representatives of real faults may be argued. Therefore, it is critical to evaluate this approach on programs that contain a reasonable number of real faults.

Moreover, the feedback mechanism as described in the problem statement section, can be added to this approach by simply making evidence nodes in BN after each test run. This may result in longer inference time, so cost-effectiveness should be carefully considered. The use of other metrics for fault-proneness and change analysis is another way of extending this work. Finally, other metrics of evaluation of prioritization techniques should be introduced and examined.

References

1. Leung, H.K.N., White, L.J.: Insights into regression testing. In: Proceedings on IEEE International Conference of Software Maintenance (ICSM). (1989) 60–69
2. Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.: Incremental regression testing. In: Proceedings of the International Conference on Software Maintenance(ICSM). (1993) 348–357
3. Chen, Y.F., Rosenblum, D.S., Vo, K.P.: Testtube: A system for selective regression testing. In: Proceedings of the ACM International Conference on Software Engineering (ICSE). (1994) 211–220
4. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology **6** (1997) 173–210
5. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering **27** (2001) 929–948
6. Wong, W.E., Horgan, J.R., London, S., Bellcore, H.A.: A study of effective regression testing in practice. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering(ISSRE). (1997) 264–274
7. Malishevsky, A.G., Rothermel, G., Elbaum, S.: Modeling the cost-benefits tradeoffs for regression testing techniques. In: Proceedings of the International Conference on Software Maintenance (ICSM). (2002) 204–213
8. Do, H., Rothermel, G., Kinneer, A.: Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. Empirical Software Engineering: An International Journal **11** (2006) 33–70
9. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering **28** (2002) 159–182
10. Kim, J.M., Porter, A.: A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the ACM International Conference on Software Engineering(ICSE). (2002) 119–129
11. Saff, D., Ernst, M.D.: Reducing wasted development time via continuous testing. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering(ISSRE). (2003) 281–292

12. Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). (2002) 97–106
13. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Timeaware test suite prioritization. In: Proceedings of the IEEE International Symposium on Software Testing and Analysis(ISSTA). (2006) 1–12
14. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
15. Okumoto, K., Goel, A.L.: Optimum release time for software systems based on reliability and cost criteria. Journal of Systems and Software **1** (1980) 315–318
16. Jensen, F.V.: Bayesian Networks and Decision Graphs. (2001)
17. Christian Collberg, Ginger Myles, M.S.: An empirical study of java bytecode programs. Technical Report TR04-11, Department of Computer Science, Univeristy of Arizona (2004)
18. Briand, L., Wüst, J.: Empirical studies of quality models in object-oriented systems. Advances in Computers **56** (2002) 98–167
19. Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering **31** (2005) 897–910
20. Elbaum, S., Gable, D., Rothermel, G.: Understanding and measuring the sources of variation in the prioritization of regression test suites. In: Proceedings of the IEEE International Symposium on Software Metrics(METRICS). (2001) 169–179
21. Chidamber, S.R., Kemerer, C.F.: Towards a metrics suite for object oriented design. In: Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA). (1991) 197–211
22. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing Co., Boston, MA, USA (1998)
23. : Apache Ant (2005) http://ant.apache.org.
24. : CKJM (2006) http://www.spinellis.gr/sw/ckjm/.
25. : Emma (2006) http://emma.sourceforge.net/.
26. : Genie/Smile (2005-2006) http://genie.sis.pitt.edu/.
27. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal **10** (2005) 405–435
28. Hema Srikanth, Laurie Williams, J.O.: System test case prioritization of new and regression test cases. In: Proceedings of the International Symposium on Empirical Software Engineering. (2005) 64–73
29. Ziv, H., Richardson, D.J.: Constructing bayesian-network models of software testing and maintenance uncertainties. In: Proceedings of the International Conference on Software Maintenance(ICSM). (1997) 100–109
30. Wooff, D., Goldstein, M., Coolen, F.: Bayesian graphical models for software testing. IEEE Transactions on Software Engineering **28** (2002) 510–525
31. Fenton, N.E., Krause, P., Neil, M.: Probability modelling for software quality control. Journal of Applied Non-Classical Logics **12** (2002) 173–188