Machine Learning Approach in Mutation Testing

Joanna Strug[1] and Barbara Strug[2]

[1] Faculty of Electrical and Computer Engineering, Cracow University of Technology
ul. Warszawska 24, 31-155 Krakow, Poland
[2] Department of Physics, Astronomy and Applied Computer Science,
Jagiellonian University, Reymonta 4, 30-059 Krakow, Poland
pestrug@cyf-kr.edu.pl, barbara.strug@uj.edu.pl

Abstract. This paper deals with an approach based on the similarity of mutants. This similarity is used to reduce the number of mutants to be executed. In order to calculate such a similarity among mutants their structure is used. Each mutant is converted into a hierarchical graph, which represents the program's flow, variables and conditions. On the basis of this graph form a special graph kernel is defined to calculate similarity among programs. It is then used to predict whether a given test would detect a mutant or not. The prediction is carried out with the help of a classification algorithm. This approach should help to lower the number of mutants which have to be executed. An experimental validation of this approach is also presented in this paper. An example of a program used in experiments is described and the results obtained, especially classification errors, are presented.

Keywords: mutation testing, machine learning, graph distance, classification, test evaluation.

1 Introduction

Software testing is a very important part of building an application. It can be also described as a process aiming at checking if the application meets the starting requirements, works as expected and satisfies the needs of all involved in.

Software testing, depending on the testing method employed, can be applied at different stages of the application development process. Traditionally most of the testing happens during the coding process and after it has been completed, but there exist approaches (for example agile), where testing is on-going. Thus the methodology of testing depends on the software development approach selected. This paper deals with mutation testing, called also mutation analysis or program mutation - a method of software testing, which involves introducing small changes in the source code (or for some programming languages byte code) of programs. Then the mutants are executed and tested by collections of tests called test suites. A test suite which does not detect mutant(s) is considered defective. The mutants are generated by using a set of mutation operators which try to mimic typical programming errors. This method aims at helping the tester assess the quality and derive effective tests.

B. Nielsen and C. Weise (Eds.): ICTSS 2012, LNCS 7641, pp. 200–214, 2012.

One of the problems with mutation testing concerns the number of mutants generated for even a small program/method, what leads to the need of compiling and executing a large number of copies of the program. This problem with mutation testing had reduced its use in practice. Over the years many tools supporting mutation testing were proposed, but reducing the number of mutants is still important aspect of mutation testing and there is a lot of research in this domain, which is briefly reviewed in the next section.

In this paper an approach based on the similarity of mutants is used. This similarity is used to reduce the number of mutants to be executed. In order to calculate such a similarity among mutants their structure is used. Each mutant is converted into a hierarchical control flow graph, which represents the program's flow, variables and conditions. On the basis of this graph form a similarity is calculate among programs. It is then used to predict whether a given test would detect a mutant or not. The prediction is carried out with the help of a classification algorithm. This approach should help to lower the number of mutants which have to be executed and at the same time help to assess quality of test suits without the need of running them. This approach shows some proximity to mutant clustering approach [8,17] as it also attempts to measure similarity of mutants, but we represent mutants in a graph form and use graph based measure rather then converting them to a special space which allows for the use of Hamming distance. Graphs have been for a long time considered to have too high computational cost to be of practical use in many domains but recently there has be a large growth of research on them, which resulted in the development of many algorithms and theoretical frameworks. Much of this research, which is briefly reviewed in the next section, deals with bio- and chemoinformatics, but some other domains were also touched upon.

The main contribution of this paper is a method to reduce the number of mutants that have to be executed in a dynamic way i.e. depending on the program for which they are generated rather than statically for a given language or the operator. Moreover this paper introduces a representation of programs that allows for comparing programs and it also proposes several measures of such a comparison. This approach was applied to two examples and the results seem to be encouraging.

The paper is organized in the following way, in the next section a related work concerning both the mutation testing and different approaches to graph analysis is briefly presented. Then, in section 3 some preliminary notion concerning classification, graphs, edit distance and graph kernels is presented, It is followed by a section 4, which presents fundamental components of our approach i.e. a hierarchical control flow graph and methods for calculating edit distance and kernel for such a graph. In section 5 experiments are presented, including the setup, and the results are discussed. Finally section 6 summarizes the paper by presenting conclusions drawn from the research presented here as well as some possible extensions, improvements and directions for future work.

2 Related Work

In this paper a number of issues from different domains is discussed. Thus in this review of related work several domains are taken into account, such as mutation testing and especially reduction approaches, different approaches to classification problem, graph analysis and, in particular, different approaches to calculating distances among graphs, (like edit distance and using kernel methods for graph data) and learning methods based on them.

Mutation testing goes back to the 70s [12] and it can be used at different stages of software development. It has also been applied to many programming languages including Java [9,10,21,22,23,24] (used in this paper). A lot of research work has also been concerned with defining mutation operators that would mimic typical errors [25]. As mentioned in the introduction one of the main problem of mutation testing is the cost of executing large number of mutants so there has been a great research effort concerning reduction costs. Two main approaches to reduction can be divided into two groups: the first containing methods attempting to reduce the number of mutants and the second - those aimed at reducing the execution costs.

One of the methods used to mutant number reduction is sampling. It was first proposed firstly by Acree [1] and Budd [6]. They still generate all possible mutants but then a percentage of these mutants is then selected randomly to be executed, and all other are discarded. Many studies of this approach were carried out, for example Wong and Mathurs [31,45] conducted an experiment using a random percentage of mutants 10% to 40% in steps of 5%.

Another approach to mutant number reduction used clustering [8,17]. It was proposed by Hussain [17] and instead of selecting mutants randomly, a subset is selected by a clustering algorithm. The process starts by generating all first order mutants, then clustering algorithm is used to put these mutants into clusters depending on the killable test cases. Mutants put into the same cluster are killed by a similar set of test cases, so a small selection of mutants is used from each cluster. All the other are then discarded.

Third approach to reduction was based on selective mutation, which consists in selecting only a subset of mutation operators thus producing smaller number of mutants [30,32]. A much wider survey of the domain of mutation testing, including approaches to reduction was carried out by Jia et al. [19].

The approach proposed in this paper is partially similar to the first two described above, as it also generates all mutants, but then only randomly selected number of them is executed and the test performance for others is assessed on the basis of their similarity to the executed mutants for which performance of test suites is thus known.

The similarity of mutants is measured using graph representation of each mutant. The use of graphs as a mean of object representation has been widely researched. They are used in engineering, system modeling and testing, bioinformatics, chemistry and other domains of science to represent objects and the relations between them or their parts. For use in computer aided design different

types of graphs were researched, not only simple ones but also hierarchical graphs (called also nested graphs [7]).

In this paper a machine learning approach based on similarity is used to analyse graphs. The need to analyze and compare graph data appeared in many domains and thus there has been a significant amount of research in this direction. Three distinctive, although partially overlapping, approaches can be noticed in the literature.

The first one is mainly based on using standard graph algorithms, like finding a maximal subgraph or mining for frequently occurring subgraphs to compare or classify graphs. The frequent pattern mining approach to graph analysis has been researched mainly in the domain of bioinformatics and chemistry [2,15,18,46,47]. The main problem with this approach is its computational cost, and a huge number of frequent substructures usually found.

The second approach is based on transforming graphs into vectors by finding some descriptive features Among others Bunke and Riesen ([4,33,34,35,36]) have done a lot of research on vector space embedding of graphs, where as features different substructures of graphs are selected. Then their number is counted in each graph and these numerical values combined in a predefined order result in a vector that captures some of the characteristics of a graph it represents. Having a graph encoded in a vector a standard statistical learning algorithms can be applied. The main problem is in finding appropriate features/substructures and in enumerating them in each graph. It usually leads to problems similar to those in frequent pattern mining (which is often used to find features counted in vector representation). Nevertheless, this approach has successfully been applied in many domains like image recognition [5], and especially the recognition of handwritten texts [26,27].

The third direction, which was proposed, among others, by Kashima and Gartner ([13,20]), is based on the theory of positive defined kernels and kernel methods [40,41]. There has been a lot of research on different kernels for structured data, including tree and graph kernels [3,13,14,20]. Tree kernels were proposed by Collins and Duffy [11] and applied to natural language processing. The basic idea is to consider all subtrees of the tree, where a subtree is defined as a connected subgraph of a tree containing either all children of a vertex or none. This kernel is computable in $O(|V_1||V_2|)$, where $|V_i|$ is the number of nodes in the $i-th$ tree [14].

In case of graph kernels there is a choice of several different ones proposed so far. One of them is based on enumerating all subgraphs of graphs G_i and calculating the number of isomorphic ones. An all subgraph kernel was shown to be NP-hard by Gartner et al [13]. Although, taking into account that in case of labelled graphs the computational time is significantly lower such a kernel is feasible in design applications. Another interesting group of graph kernels is based on computing random walks on both graphs. It includes the product graph kernel [13] and the marginalized kernels [20]. In product graph kernel a number of common walks in two graphs is counted. The marginalized kernel on the other hand is defined as the expectation of a kernel over all pairs of label sequences

from two graphs. These kernels are computable in polynomial time, ($O(n^6)$ [14]), although for small graphs it may be worse then 2^n, when the neglected constant factors contribute stronger.

The main research focus is on finding faster algorithms to compute kernels for simple graphs, mainly in bio- and chemoinformatics. Yet, to author's best knowledge, no research has been done in the area of defining and testing kernels for different types of graphs, such as hierarchical control flow graphs proposed in this paper..

3 Preliminaries

Classification is one of main tasks being part of machine learning. It consists in identifying to which of a given set of classes a new element (often called observation) belongs. This decision is based on a so called training set, which contains data about other elements (often called instances) whose class membership is known. The elements to be classified are analysed on the basis of their properties, called features. These features can be of different types (categorical, ordinal, integer-valued or real-valued), but some known algorithms work only if the data is real-valued or integer-valued based. An algorithm which implements classification is known as a classifier. In machine learning, classification task is considered to be a supervised learning algorithm, i.e. learning process uses a training set of elements classified correctly.

There is a number of known classification algorithms. One of them is $k-NN$ (k nearest neighbours, where k is a parameter), used in this paper. In $k-NN$ classifier training set consists of vectors in a multidimensional space, for which a class membership is known. Thus training stage of the classifier consists only in storing the vectors and class labels of the elements of the training set. Then, during the actual classification for elements of unknown class membership, a distance from the new element to all elements of the training set is calculated and it is assigned to the class which is most frequent among the k training examples nearest to that new one.

In majority of known classification algorithms, including $k-NN$ an instance to classify is described by a feature vector containing properties of this instance. As in this paper graphs are used, not vectors, to represent objects to classify a way of calculating distance between two graphs is needed. Two such methods, graph edit distance and graph kernel, are briefly presented in the following, together with some basic notions. Then, in the next section, we show how these concepts can be extended to deal with hierarchical flow graphs proposed in this paper.

3.1 Graphs

A simple graph G is a set of nodes (called also vertices) V and edges E, where $E \subset V^2$. Each node and edge can be labeled by a function ξ, which assigns labels to nodes and edges. A walk w of length $k-1$ in a graph is a sequence of nodes $w = (v_1, v_2, \ldots, v_k)$ where $(v_i, v_j) \in E$ for $1 \leq i, j \leq k$. If $v_i \neq v_j$ for $i \neq j$ then a walk w is called a *path*.

Graph Edit Distance. A graph edit distance (GED) approach is based on the fact that a graph can be transformed to another one by performing a finite number of graph edit operations which may be defined in a different way, depending on algorithms. GED is then defined as the least-cost sequence of such edit operations. Typically edit operation sequences include node edge insertion,node and edge deletion, node and edge substitution (label change). A cost function has to be defined for each of the operations and the cost for the edit operation sequence is defined as the sum of costs for all operations present in a given sequence. It has to be noticed that the sequence of edit operations and thus the cost of the transformation of a graph into another one is not necessary unique, but the lowest cost is and it is used as GED. For any given domain of application the two main issues are thus the way in which the similarity of atoms (nodes and edges) is defined and what is the cost of each operation. For labelled graphs, thus having labels for nodes, edges, or both of them, the deletion/insertion/substitution costs in the GED computations may depend on these labels.

Graph Kernels. Another approach to use traditional classification algorithms for non vector data is based on the so called kernel trick, which consists in mapping elements from a given set A into an inner product space S (having a natural norm), without ever having to actually compute the mapping,i.e. graphs do not have to be mapped into some objects in space S, only the way of calculation the inner product in that space has to be well defined. Linear classifications in target space are then equivalent with classifications in source space A. The trick allowing to avoid the actual mapping consists in using the learning algorithms needing only inner products between the elements (vectors) in target space, and defining the mapping in such a way that these inner products can be computed on the objects in the source the original space by means of a kernel function. For the classifiers a kernel matrix K must be positive semi-definite (PSD), although there are empirical results showing that some kernels not satisfy this requirement may still do reasonably well, if a kernel well approximates the intuitive idea of similarity among given objects. Formally a positive definite kernel on a space X is a symmetric function $K : X^2 \to \mathbf{R}$, which satisfies $\sum_{i,j=1}^{n} a_i a_j K(x_i, x_j) \geq 0$, for any points $x_1, \ldots, x_n \in X$ and coefficients $a_1, \ldots, a_n \in \mathbf{R}$.

The first approach of defining kernels for graphs was based on comparing all subgraphs of two graphs. The value of such a kernel usually equals to the number of identical subgraphs. While this is a good similarity measure, the enumeration of all subgraphs is a costly process. Another approach is based on comparing all paths in both graphs. It was used by Kashima [20] who proposed the following equation:

$$K(G_1, G_2) = \sum_{path_1, path_2 \in V_1^* \times V_2^*} p_1(path_1) p_2(path_2) K_L(lab(path_1), lab(path_2)), \tag{1}$$

where p_i is a probability distribution on V_i^*, and K_L is a kernel on sequences of labels of nodes and edges along the path $path_i$. It is usually defined as a product of subsequent edge and node kernels. This equation can be seen as a marginalized

kernel and thus is a positive defined kernel [43]. Although computing this kernel requires summing over an infinite number of paths it can be done efficiently by using the product graph and matrix inversion [13]. Another approach uses convolution kernels [16], which are a general method for structured data (and thus very usefull for graphs). Convolution kernels are based on the assumption that structured object can be decomposed into components, then kernels are defined for those components and the final kernel is calculated over all possible decompositions.

4 Data Preparation

To carry out an experiment a number of steps was needed to prepare the data. Firstly, two relatively simple, but nevertheless representative, examples were selected and mutants for them were generated by using Mujava tool [29]. One of the examples was a simple search presented in Fig. 1, For this example Mujava generated 38 mutants; for the second examples there were 87 mutants. The mutants were then converted into graph form described below.

```
public int search(int v){
int i;
for(i=0;i<size;i=i+1)
if(values[i]==v) return i;
return -1;
}
```

```
public int search(int v){
    int i;
    for(i=0;++i<size;i=i+1)
    if(values[i]==v) return i;
    return -1;
    }
```

Fig. 1. A simple search method and one of its AOIS (Arithmetic Operator Insertion [29]) mutants

4.1 Hierarchical Control Flow Graphs

Although a well known method of representing programs or their components (methods) is a control flow diagram (CFD), it cannot be directly used to compare programs, as we need to compare each element of any expression or condition separately and a traditional CFD labels its elements by whole expressions. So in this paper a combination of CFD and hierarchical graphs is proposed. It adds a hierarchy to this diagram enabling us to represent each element of a program in a single node and thus making the graphs more adequate to comparison. An example of such a hierarchical control flow graph (HCFG) is depicted in Figs. 2a and b. It represents a method $Search(...)$ and its mutant depicted in Fig. 1a and b, respectively. It can be noticed that the insertion of ++ into variable i in a for loop is represented by an appropriate expression tree replacing a simple node labelled i inside node labelled for.

Let for the rest of this paper R_V and R_E be the sets of node and edge labels, respectively. Let ϵ be a special symbol used for unlabelled edges. The set of node labels consists of the set of all possible keywords, names of variables, operators, numbers and some additional grouping labels (like for example *declare* or *array* shown in Fig. 2. The set of edge labels contains Y and N.

Definition 1. *(Labelled hierarchical control flow graph) A labelled hierarchical control flow graph HCFG is defined as a 5-tuple* (V, E, ξ_V, ξ_E, ch) *where:*

1. V *is a set of nodes,*
2. E *is a set of edges,* $E \subset V \times V$ *,*
3. $\xi_V : V \to R_V$ *is a node labelling function,*
4. $\xi_E : E \to R_E \cup \{\epsilon\}$ *is an edge labelling function,*
5. $ch : V \to P(V)$ *is a function assigning to each node a set of its children, i.e. nodes directly nested in* v*.*

Let, for the rest of this paper, $ch(v)$ denotes the set of children of v, and $|ch(v)|$ the size of this set. Let *anc* be a function assigning to each node its ancestor and let λ be a special empty symbol (different from ϵ),$anc : V \to V \cup \{\lambda\}$, such that $anc(v) = w$ if $v \in ch(w)$ and λ otherwise.

4.2 Hierarchical Control Flow Graphs Distance

HCFG Edit Distance. To define edit cost for a particular graph a cost function for edit operations must be defined. In case of HCF graphs it was defined to mimic as much as possible the influence of a given operation over the similarity. Costs for changing labels were set separately for all pairs of possible keywords, variable names and operators. For example cost of changing the operator in a condition from $<$ into $<=$ is lower than changing $==$ into $!=$ as the perceived difference between them is higher. Changing the conditional expression into arbitrary *true* or $false$ will be even higher, and it is well represented in the edit distance concept as replacing the expression tree with a single node requires significantly more delete/insert operations.

HCFG Kernel. The edit distance does not take into account the additional information contained in the hierarchical structure HCFG. To incorporate this information into similarity calculations a hierarchical substructure kernel K_{HCFG} is proposed in this paper. It takes into account the label of a given node, number of its children (and thus the internal complexity), the label of its hierarchical ancestor (and thus its position within the structure of the program), and the number and labels of edges connecting this node with its neighbourhood nodes (both incoming and outgoing edges are taken into account) This substructure kernel uses node, edge and tree kernels. The node and edge kernels are defined below. The tree kernel, used within the node one to compare expression trees, is a standard one [11].

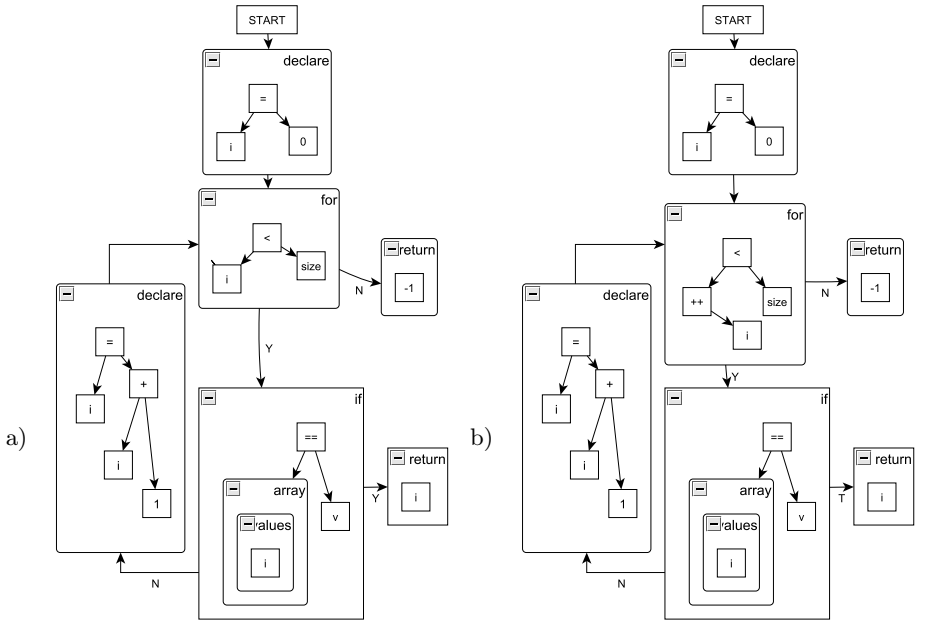

Fig. 2. Examples of flow graphs a) a graph for program from Fig. 1a, b) a flow graph for one of AOIS mutants (from Fig. 1b)

Definition 2. *A node kernel, denoted $k_V(v, w)$, where v, and w are nodes of a hierarchical control flow graph, is defined in the following way:*

$$k_V(v,w) = \begin{cases} 1 & : \xi_V(v) = \xi_V(w) \wedge |ch(v)| = |ch(w)| = 0 \\ k_V(ch(v), ch(w)) & : \xi_V(v) = \xi_V(w) \wedge |ch(v)| = |ch(w)| = 1 \\ K_T(ch(v), ch(w)) & : |ch(v)| > 1 \vee |ch(w)| > 1 \\ 0 & : \xi_V(v) \neq \xi_V(w). \end{cases}$$

It can be observed that for nodes having more than one child, thus containing an expression tree, a tree kernel K_T is used to compute the actual similarity. For nodes having different labels the kernel returns 0, while for nodes containing one children the node kernel is called recursively.

Definition 3. *An edge kernel, denoted $k_E(e_i, e_j)$, where e_i, and e_j are edges of a hierarchical flow graph, is defined in the following way:*

$$k_E(e_i,e_j) = \begin{cases} 1 : \xi_E(e_i) = \xi_E(e_j) \\ 0 : \xi_V(e_i) \neq \xi_V(e_j). \end{cases}$$

On the basis of the above kernel a similarity for HCFG is computed.

Definition 4.

$$K_{HCFG}(G_i, G_j) = \sum_{i=1}^{m} \sum_{j=1}^{n} K_S(S_i, S_j), \tag{2}$$

where m and n, is the number of hierarchical nodes in each graph and

$$K_S(S_i, S_j) = k_{node}(v_i, v_j) + k_{node}(anc(v_i), anc(v_j)) + \sum_{r=1}^{C_n} \sum_{t=1}^{C_m} k_{node}(c_r(v_i), c_t(v_j)) + \sum_{w_j \in Nb(v_j)} \sum_{w_j \in Nb(v_j)} k_{edge}((v_i, w_i), (v_j, w_j)) k_{node}(w_i, w_j), \quad (3)$$

where each S_i is a substructure of G_i consisting of node v_i, its direct ancestor $anc(v_i)$, all its children $ch(v_i)$ (where with C_n is the number of children and $c_n(v_i)$ - the $n-th$ child of v_i) , and its neighbourhood $Nb(v_i)$.

This kernel is based on the decomposition of a graph into substructures according to the concept of $R-convolution$ kernels and thus is positive semidefinite [16], and so acceptable as a kernel function [40].

Remark on Computational Costs. Both edit distance and graph kernel are known to have a high computational cost, what was mentioned in sections 1 and 2. But in case of HCFG we have a special situation, i.e. as each graph represents a first order mutant, any two graphs can differ in at most two places. Moreover we know a priori where the change happened, and all the remaining elements of both graphs are identical. As a result the actual computation of both edit distance and HCFG kernel can be done much more efficiently than in general case of two arbitrarily chosen graphs.

5 Experiments and Results

For each set of mutants a k-NN classification algorithm was run using two different distance measures, an edit distance and a distance computed from HCFG kernel. For the first example three test suites were used and the set of mutants was randomly divided into three parts of similar size, the first was used as a training set and the other as instances to classify. The classification was then repeated using subsequent subsets as training sets. The whole process was repeated five times using different partitions of the set of mutants and the results obtained were averaged. Table 1 presents the results obtained for this example using HCFG edit distance to compute distances in $k-NN$ classifier. Parameter k was, after some experimental tuning, set to 5 for all experiments. The first column of the table shows the percentage of instances classified correctly. The results for mutants classified incorrectly are presented separately for those classified as detectable, while actually they are not (column labelled incorrect killed) and for those classified as not detected, while they actually are detected by a given test suite (column labelled incorrect not killed). Calculating these results separately was motivated by the meaning of these misclassifications. While classifying a mutant as not detected leads to overtesting, the misclassification of the second type can result in missing some errors in code, what is more dangerous. As the results are also used to evaluate the quality of test suites used, incorrectly

classifying a mutant as not detected leads to giving a test suite lower score than actual one, why the second misclassification leads to overvaluation of a given test suite. Again, while the first situation is surely not desired, the second one poses more problems, especially as it may lead to a situation when a mutant not detected by any test suite would be labelled as detected, thus resulting in undetected errors in code.

It can be observed that the classification performed reasonably well for all test suits, with the exception of TS1. Deeper analysis of this case seems to suggest that it results from the random partition of the set of mutants for this test suite in which the training set contained unproportionally large number of undetectable mutants. This situation also suggests to perform the partition of mutants in a "smarter" way instead of random. One possible way to do it is to select proportional number of mutants of each type (generated by a given type of mutation operators). The results obtained with the use of HCFG kernel, presented in Table 2, are slightly better in general, especially the classification for TS1 improved significantly, although it may be due to better choice of training sets. It can be also noticed that, while the percentage of correctly classified mutants for test suite 3 is a bit lower, (but the difference is small), less mutants were incorrectly classified as detectable, although this gain happened at the expense if larger classification error in the last column. The results show that the classification improvements for the kernel method are not very significant, but more experiments are needed to decide whether this approach is worth its slightly higher computational cost.

Table 1. The classification of mutants of example 1 with the use of GED

	correct	incorrect killed	incorrect not killed
TS 1	65.2%	13.06%	21.74%
TS 2	78.25%	8.7%	13.5%
TS 3	82.6%	8.7%	8.7%

Table 2. The classification of mutants of example 1 with the use of kernel

	correct	incorrect killed	incorrect not killed
TS 1	75.55%	5.45%	19.00%
TS 2	84.1%	6.65%	9.25%
TS 3	82.2%	4.7%	12.7%

For the second example five test suites were used and, as there were more mutants, their set was divided into four parts of similar size, with, like in first example, the first part being used as a training set and the others as instances to classify. The classification was then repeated using subsequent subsets as training sets. The whole process was also repeated five times with different partitions of the set of mutants and the results obtained were averaged. Table 3 presents the results obtained with the use of edit distance and Table 4 - with the use of

kernel based distance. Similarly to the first example the kernel based approach has produced slightly better results for correct classifications, with the exception of TS 2, where the error is slightly higher, but only by 0.2%. However a slightly higher improvement can be observed in having a lower percentage of mutants incorrectly classified as detectable. It can also be noticed that the results for TS 3 were visibly worse than for other suits. Closer inspection seems to suggest that this is also a problem with randomly partitioning set of mutants. As TS 2 detects only 22 out of 87 mutants there may occur an over representation of detectable mutants in the training set thus leading to incorrectly classifying many mutants as detectable. As in the first example it suggests replacing random partitioning by another one. Here a useful idea seems to be selecting into training set mutants in such a way that would preserve the proportion of both detectable and undetectable mutants close to the one in the whole set.

Table 3. The classification of mutants of example 2 with the use of GED

	correct	incorrect killed	incorrect not killed
TS 1	75.7%	12.1%	12.2%
TS 2	73.4%	6.5%	20.1%
TS 3	60.5%	26.2%	16.3%
TS 4	78.2%	10.3%	11.5%
TS 5	76.4%	11.3%	12.3%

Table 4. The classification of mutants of example 2 with the use of kernel

	correct	incorrect killed	incorrect not killed
TS 1	79.1%	6.3%	14.6%
TS 2	73.2%	4.5%	22.3%
TS 3	61.5%	22.6%	20.9%
TS 4	85.1%	4.6%	10.5%
TS 5	79.2%	9.53%	11.3%

6 Conclusions and Future Work

In this paper an approach to classification of mutants was proposed as a tool to reduce the number of mutants to be executed and to evaluate the quality of test suits without executing them against all possible mutants. This method deals with reducing the number of mutants that have to by executed in a dynamic way i.e. depending on the program for which they are generated rather than statically for a given language or the operator.The approach needs still more experiments to fully confirm its validity, but the results obtained so far are encouraging.

However, several problems were noticed during the experiment that require further research. Firstly, a random selection, although performing reasonably

well, causes problems for some test suits. Possible solutions, as suggested in the discussion of results, include selecting mutants to assure they represent diversity of mutation operations thus avoiding selecting to the training set mutants generated by the same type of operations. The second solution is to select a number of detectable and undetectable mutants to preserve proportions from the full set. We plan to investigate both approaches to check whether they improve results in a significant way.

Another direction for future research is connected with the use of kernels. To make better use of them one of kernel based classifiers, for example support vector machines, could be used instead of $k - NN$. The kernel itself also offers some possibilities for improvements. The node kernel proposed in this paper is based on the label of the node independently form its position ("depth") in the hierarchy; adding some factor proportional to the depth of the node is also planned to be researched.

References

1. Acree, A.T.: On Mutation, PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia (1980)
2. Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD 1993), Washington, DC, pp. 207–216 (1993)
3. Borgwardt, K.M., Kriegel, H.P.: Shortest-path kernels on graphs. In: ICDM 2005, pp. 74–81 (2005)
4. Bunke, H., Riesen, K.: Improving vector space embedding of graphs through feature selection algorithms. Pattern Recognition 44(9), 1928–1940 (2011)
5. Bunke, H., Riesen, K.: Recent advances in graph-based pattern recognition with applications in document analysis. Pattern Recognition 44(5), 1057–1067 (2011)
6. Budd, T.A.: Mutation Analysis of Program Test Data. PhD Thesis. Yale University, New Haven, Connecticut (1980)
7. Chein, M., Mugnier, M.L., Simonet, G.: Nested Graphs: A Graph-based Knowledge Representation Model with FOL Semantics. In: Proceedings of the 6th International Conference "Principles of Knowledge Representation and Reasoning" (KR 1998), Trento, Italy, pp. 524–534. Morgan Kaufmann Publishers (June 1998)
8. Ji, C., Chen, Z., Xu, B., Zhao, Z.: A Novel Method of Mutation Clustering Based on Domain Analysis. In: Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009), July 1-3. Knowledge Systems Institute Graduate School, Boston (2009)
9. Chevalley, P.: Applying Mutation Analysis for Object-oriented Programs Using a Reflective Approach. In: Proceedings of the 8th Asia- Pacific Software Engineering Conference (APSEC 2001), Macau, China, December 4-7, p. 267 (2001)
10. Chevalley, P., Th'evenod-Fosse, P.: A Mutation Analysis Tool for Java Programs. International Journal on Software Tools for Technology Transfer 5(1), 90–103 (2002)
11. Collins, M., Duffy, N.: New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. In: Proceedings of ACL 2002 (2002)

12. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. Computer 11(4), 34–41 (1978)
13. Gartner, T.: A survey of kernels for structured data. SIGKDD Explorations 5(1), 49–58 (2003)
14. Gartner, T.: Kernels for structured data. Series in Machine Perception and Artificial Intelligence. World Scientific (2009)
15. Han, J., Pei, J., Yin, Y., Mao, R.: Mining Frequent Patterns without Candidate Generation: A Frequent-pattern Tree Approach. Data Mining and Knowledge Discovery: An International Journal 8(1), 53–87 (2004)
16. Haussler, D.: Convolutional kernels on discrete structures. Technical Report UCSC-CRL-99-10, Computer Science Department, UC Santa Cruz (1999)
17. Hussain, S.: Mutation Clustering, Masters Thesis. King's College London, Strand, London (2008)
18. Inokuchi, A., Washio, T., Motoda, H.: An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 13–23. Springer, Heidelberg (2000)
19. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Trans. Software Eng., 649–678 (2011)
20. Kashima, H., Tsuda, K., Inokuchi, A.: Marginalized Kernels Between Labeled Graphs. In: ICML 2003, pp. 321–328 (2003)
21. Kim, S., Clark, J.A., McDermid, J.A.: Assessing Test Set Adequacy for Object Oriented Programs Using Class Mutation. In: Proceedings of the 3rd Symposium on Software Technology (SoST 1999), Buenos Aires, Argentina, September 8-9 (1999)
22. Kim, S., Clark, J.A., McDermid, J.A.: The Rigorous Generation of Java Mutation Operators Using HAZOP. In: Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 1999), Paris, France, November 29-December 1 (1999)
23. Kim, S., Clark, J.A., McDermid, J.A.: Class Mutation: Mutation Testing for Object-oriented Programs. In: Proceedings of the Net. Object Days Conference on Object-Oriented Software Systems (2000)
24. Kim, S., Clark, J.A., McDermid, J.A.: Investigating the effectiveness of object-oriented testing strategies using the mutation method. In: Proceedings of the 1st Workshop on Mutation Analysis (MUTATION 2000), Published in Book Form, as Mutation Testing for the New Century, San Jose, California, October 6-7, pp. 207–225 (2001)
25. King, K.N., Offutt, A.J.: A Fortran Language System for Mutation- Based Software Testing. Software: Practice and Experience 21(7), 685–718 (1991)
26. Liwicki, M., Bunke, H., Pittman, J.A., Knerr, S.: Combining diverse systems for handwritten text line recognition. Mach. Vis. Appl. 22(1), 39–51 (2011)
27. Liwicki, M., Schlapbach, A., Bunke, H.: Automatic gender detection using on-line and off-line information. Pattern Anal. Appl. 14(1), 87–92 (2011)
28. Preller, A., Mugnier, M.-L., Chein, M.: Logic for Nested Graphs. Computational Intelligence 14(3), 335–357 (1998)
29. Ma, Y., Offutt, J., Kwon, Y.R.: MuJava: a mutation system for java. In: ICSE, pp. 827–830 (2006)
30. Mathur, A.P.: Performance, Effectiveness, and Reliability Issues in Software Testing. In: Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC 1979), Tokyo, Japan, September 11-13, pp. 604–605 (1991)

31. Mathur, A.P., Wong, W.E.: An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria, Purdue University, West Lafayette, Indiana, Technique Report (1993)
32. Offutt, A.J., Rothermel, G., Zapf, C.: An Experimental Evaluation of Selective Mutation. In: Proceedings of the 15th International Conference on Software Engineering (ICSE 1993), pp. 100–107. IEEE Computer Society Press, Baltimore (1993)
33. Richiardi, J., Van De Ville, D., Riesen, K., Bunke, H.: Vector Space Embedding of Undirected Graphs with Fixed-cardinality Vertex Sequences for Classification. In: ICPR 2010, pp. 902–905 (2010)
34. Riesen, K., Bunke, H.: Cluster Ensembles Based on Vector Space Embeddings of Graphs. In: Benediktsson, J.A., Kittler, J., Roli, F. (eds.) MCS 2009. LNCS, vol. 5519, pp. 211–221. Springer, Heidelberg (2009)
35. Riesen, K., Bunke, H.: Dissimilarity Based Vector Space Embedding of Graphs Using Prototype Reduction Schemes. In: Perner, P. (ed.) MLDM 2009. LNCS, vol. 5632, pp. 617–631. Springer, Heidelberg (2009)
36. Riesen, K., Bunke, H.: Reducing the dimensionality of dissimilarity space embedding graph kernels. Eng. Appl. of AI 22(1), 48–56 (2009)
37. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph. Transformations. Fundations, vol. 1. World Scientific, London (1997)
38. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph. Transformations. Applications, Languages and Tools, vol. 2. World Scientific, London (1999)
39. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. 1-462 (2004)
40. Schölkopf, B., Smola, A.J.: A Short Introduction to Learning with Kernels. In: Mendelson, S., Smola, A.J. (eds.) Advanced Lectures on Machine Learning. LNCS (LNAI), vol. 2600, pp. 41–64. Springer, Heidelberg (2003)
41. Schölkopf, B., Smola, A.J.: Learning with kernels. MIT Press, Cambridge (2002)
42. Strug, B.: Using Kernels on Hierarchical Graphs in Automatic Classification of Designs. In: Jiang, X., Ferrer, M., Torsello, A. (eds.) GbRPR 2011. LNCS, vol. 6658, pp. 335–344. Springer, Heidelberg (2011)
43. Tsuda, K., Kin, T., Asai, K.: Marginalized kernels for biological sequences. Bioinformatics 18, 268–275
44. Vishwanathan, S.V.N., Borgwardt, K.M., Schraudolph, N.N.: Fast Computation of Graph Kernels. In: NIPS 2006, pp. 1449–1456 (2006)
45. Wong, W.E.: On Mutation and Data Flow. PhD Thesis, Purdue University, West Lafayette, Indiana (1993)
46. Yan, X., Yu, P.S., Han, J.: Substructure Similarity Search in Graph Databases. In: Proc. of 2005 Int. Conf. on Management of Data, SIGMOD 2005 (2005)
47. Yan, X., Yu, P.S., Han, J.: Graph Indexing: A Frequent Structure-based Approach. In: Proc. of 2004 Int. Conf. on Management of Data, SIGMOD 2004 (2004)