Using Decision Trees to aid Algorithm Selection in Combinatorial Interaction Tests Generation

Angelo Gargantini
Dip. di Ingegneria
University of Bergamo-Italy
Email: angelo.gargantini@unibg.it

Paolo Vavassori
Dip. di Ingegneria
University of Bergamo-Italy
Email: paolo.vavassori@unibg.it

***Abstract*—It has been widely observed that there is no a single *best* CIT generation algorithm; instead, different algorithms perform best in terms of test suite size and time, also depending on different combinatorial models. Rather than following the traditional approach of leaving the choice of the best generator for a given class of models and for given testing requirements to the user, we want to automate the algorithm selection process among a given set of techniques (called *portfolio*). The proposed approach takes as input a distribution of combinatorial models and their test suites generated using several tools, then, using data-mining techniques, it permits to predict the algorithm that performs better given the cost estimated to execute a single test and the model characteristics. As predictors, we decide to use *decision trees* because they have been one of the most widely used decision support tool for many years. Their attraction lies in the simplicity of the resulting model, where a decision tree (at least one that is not too large) is quite easy to view, understand, and, importantly, explain even if it may not always deliver the best performances. We demonstrate the effectiveness of our approach to automated algorithm selection in extensive experimental results on data sets including models commonly presented in literature.**

I. Introduction

During the last years, the combinatorial interaction testing (CIT) community has proposed many approaches for solving combinatorial testing problems. New techniques, tools, and algorithms are continuously proposed, benchmarked, and proved to improve over the state of the art in many cases. It remains unclear which approach must be considered as the *best*: it seems even impossible to state that a certain tool/technique clearly outperforms the others. In fact, even if we consider only the two main families of generation algorithms, namely greedy techniques and meta-heuristic approaches, we cannot say that one is surely better than the other. It is well known that greedy techniques are faster because they perform every decision only once while meta-heuristic ones may revisit their choices. It is reasonable to postulate that greedy algorithms run faster but meta-heuristic searches produce smaller samples size. Although recent works focus their attention to the improvement of both these techniques in order to fill their gaps in terms of performance, the contrast, between the size reduction of samples and the generation time reduction, is sharp. The reason lies in the nature of the problem: CIT is clearly a multi-objective problem where optimal decisions need to be taken in the presence of two conflicting objectives: namely generation time versus test suite size. Depending on how much the tester is willing to wait and on the cost of executing a single test, one tool or another may be the best choice. Moreover, also the features of the model under test may influence the choice of the tool: a tool may perform very well for small models but have problems of scalability. In this paper, we use a mathematical model that formalizes the goal of minimizing the testing costs.

It would be good that the tester could choose the right algorithm among many. With this intent, we originally devised CitLab, an open framework for combinatorial testing [1], [2]. It allows CIT researchers to share their models and to run the major CIT tools (if supported) into a common environment for a scrupulous performance comparison. CitLab supports already several test generation tools and new one may be added in the future. However, the choice of the right algorithm to use is still left to the testers. In this paper, we try to introduce an automatic component that can suggest the user the right choice depending on several inputs (cost of executing a single test, characteristics of the model, and so on).

Our approach tries to solve a classical selection problem of one algorithm in an *algorithm portfolio*. This idea was originally presented by Huberman et al. [3] to describe the strategy of running several algorithms in parallel, potentially with different algorithms being assigned different amounts of CPU time. Several authors have since used the term in a broader way that encompasses any strategy that leverages multiple black-box algorithms to solve a single problem instance.

In this paper, we use the term *portfolio* to describe a set of algorithms from which to choose the one or ones best for the model under test. Namely, the algorithms for CIT test generations supported in CitLab are ACTS [4], [5], CASA [6], [7], and MEDICI [8].

We propose to use as tool for supporting the user in the decision of the technique to be used for test generation, decision trees, which can guide in a simple way the users in the best choice. We present the process of building such decision trees by using a data mining process that starts from the analysis of the performance of CASA, ACTS and MEDICI over a selection of 114 models. The ideal solution to the

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
4th International Workshop on Combinatorial Testing (IWCT 2015)
978-1-4799-1885-0/15/$31.00

algorithm selection problem would be to consult an oracle that knows exactly the amount of time that each algorithm would take to solve a given problem instance and the size of the test suite that would be generated, and then to select the algorithm with the best performance. In the context of the CIT, knowing the exact time and the exact test suite size is almost impossible without actually executing the tool itself. Moreover, in general, there is no need to know the size of and the time since the tester suffices to know which tool to execute in order to achieve a certain goal (for example a very small test suite size). For these reasons, our decision trees will learn which tool to select for a particular model under test. Experiments show that using decision trees can significantly reduce the cost of testing w.r.t. using a fixed tool for all the testing generation tasks.

The paper is organized as follows. In Section II, we present some background: the cost model, the decision trees, and CITLAB. The process of building a decision tree is presented in Section III. The experimental results that prove the efficacy of our approach (together with some threats to validity) are reported in Section IV. Section VI presents some related work and Sect. VII gives some future directions of our work. Sect. VIII concludes the paper.

II. BACKGROUND

A. Defining the best combinatorial generation algorithm

There are many algorithms and tools for combinatorial test generation. In a recent book [9], Zhang et al. have counted 12 tools/framework actively maintained for CIT testing, while the pairwise web site[1] lists around 39 tools. Another recent survey lists around 50 papers dealing with test generation [10]. The research community and some commercial activities continuously propose new algorithms (or improved version of existing techniques) and software for CIT test generation. It is apparent that the choice of the right tool for test generator can be difficult. Even if one wants to focus on one single tool, it may have several options that make even its usage not an easy task. To simplify the problem we can limit our attention to free tools, consider models containing constraints, ignore other aspects like usability, interoperability and so on, and focus only on the *test generation time* and *test suite size*. Even in this case the identification of the best test generator is not easy because the test generation for CIT is a typical multi objective problem in which test suite size and test generation time are two conflicting objectives: a tool can be very fast but produce enormous test suites, while another may guarantee to find very small complete test suite but require hours of computation.

In order to allow a fair comparison among tools, to guide the choice of the best suitable one, and to devise a technique to extract the decision tree that can help the user in the choice, we first borrow the model proposed in [6] for roughly estimating the cost of testing ($cost$) as the total time for test generation ($time_{gen}$) plus test execution time, which depends on the size of the test suite ($size$) and on the time necessary to execute every single test ($time_{test}$):

$$cost = time_{total} = time_{gen} + size \times time_{test} \quad (1)$$

In this model, we assume that the cost of testing (both test generation and execution) is equal to the time required for the activity. In case the cost is linearly bound to the time, for instance, because testing requires the use of a dedicated server that has an hourly cost, or because tests are manually executed by a person with a temporal cost, our model still holds (provided that a constant is introduced). We leave as future work the study of other models of costs. The cost of executing one single test is also called unitary execution cost (**UEC**).

B. CITLAB

In order to enable a real choice between several algorithms and to avoid a vendor lock-in, the tester should have a simple syntax for combinatorial problems and a framework in which several generation techniques can be inserted as plug-ins. This has been the main idea behind our framework called CITLAB [1], [2]. The CITLAB allows importing/exporting models of combinatorial problems from/to different application domains, by means of a common interchange syntax notation and a corresponding inter-operable semantic meta-model. Moreover, the tool is a framework allowing embedding and transparent invocation of multiple, different implementations of combinatorial algorithms. CITLAB has been designed tightly integrated with the Eclipse IDE framework, by means of its plug-in extension mechanism. CITLAB already supports three main generation techniques: ACTS [4], [5], CASA [6], [7], and MEDICI [8]. ACTS is a tool developed by the NIST and implements several variants of the In Parameter Order (IPO) strategy. CASA is a tool developed at the University of Nebraska and it is based on simulated annealing, a well-studied meta-heuristic algorithm. MEDICI is a tool based on the use of Multivalued Decision Diagrams (MDDs) and it includes a novel variation of the classical greedy policy weighting the compatibility among the tuples. All three support constraints and they are freely available. ACTS and CASA have a large user base and they are very often used in comparison studies. Using CITLAB allows us to perform all the experiments in a very controlled environment on the same computer and using exactly the same examples. We assume in this paper that the CIT portfolio is constituted by the three tools mentioned above. New ones could be added in the future.

C. Decision Tree

We propose the use of decision trees for the suggestion of right tool for test generator. Decision trees (also referred to as classification and regression trees) are the traditional building blocks of data mining and the classic machine learning algorithm. Since their development in the 1980s, decision trees have been the most widely deployed machine-learning based data mining model builder. Their attraction lies in the simplicity of the resulting model, where a decision tree (at

[1] http://www.pairwise.org

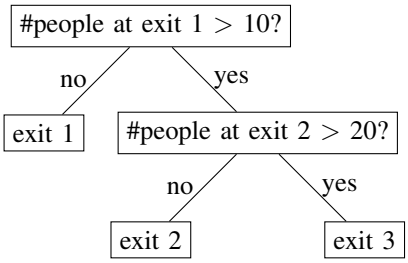

Figure 1: A decision tree representing the exit selection in a crowded parking area.

least one that is not too large) is quite easy to view, understand, and, importantly, explain. Classification tree structure is used in many different fields, such as medicine, logic, problem solving, and management science. It is also a traditional computer science structure for organizing data.

Fig. 1 shows a simple decision tree that can be used to decide the exit in a parking area depending on the number of people in it.

D. Tools for Data Mining

For data mining, we use the free and open source software Rattle [11], built on top of the R statistical software package [12]. Rattle has been developed using the Gnome toolkit with the Glade graphical user interface (GUI) builder. Rattle provides considerable data mining functionality by exposing the power of the R Statistical Software through a graphical user interface. There is a Log Code tab, which replicates the R code for any activity undertaken in the GUI, which can be copied and pasted. Rattle can be used for statistical analysis, or model generation and it allows the partition of the dataset into *training, validation, and testing* subsets.

We consider R one of the most comprehensive statistical analysis package available. It incorporates all of the standard statistical tests, models, and analyses, as well as providing a comprehensive language for managing and manipulating data. New technologies and ideas often appear first in R. Rattle (the R Analytical Tool To Learn Easily) provides a simple and logical interface for data mining. The application runs under GNU/Linux and Windows. We choose Rattle because it provides an intuitive interface that takes the practitioner through the basic steps of data mining.

III. Process of Building the Decision Tree

In this section, we explain the process of building a decision maker for the selection of the best algorithm for combinatorial test generation. With *decision maker*, we mean a statistical predictor that is able to forecast which generator produces the minimum total test cost for a specific model. The predictor will need some data as inputs, like the cost of execution of a single test and some of the features of the model, and will produce as suggestion an algorithm.

The process of finding such decision maker is a typical data-mining problem. The CRISP-DM (Cross Industry Standard Process for Data Mining) [13] identifies five steps within a typical datamining project:

1) Problem Understanding
2) Data Understanding and Preparation
3) Modeling
4) Evaluation
5) Deployment

A. Problem Understanding

During this phase, we try to understand the project objectives and requirements, and then identify the data that define the problem.

The main objective of this project is to find a decision tree that can help the user to choose the right CIT generator that minimizes the cost defined in Equation 1. We want to devise a predictor that given an estimated cost for each single test ($time_{test}$) and a certain model, is able to suggest a test generator that minimizes the final cost. Note that the time required for each test generation ($time_{gen}$) and the *size* of the produced test suite depend on both the chosen test generator and on the attributes of the combinatorial model. The cost for a single test $time_{test}$ plays a very important role in the selection of a test generator, because for small values of $time_{test}$ the cost is mainly due to the $time_{gen}$, while for big values of $time_{test}$, the $size$ is more important. Since $time_{gen}$ and $size$ depend on the generator and they are generally negatively correlated, the $time_{test}$ greatly influences the choice of the best test generator. So the first data that influence our problem is:

- **UEC**: the unitary execution cost. This is given by the tester as an average of the expected time required to execute a single test.

Regarding the model attributes that influence the test generation; we can identify the following possible candidates:

- **N.var**: number of variables.
- **N.constraints**: number of constraints. In this case, we can normalize the number of constraints by converting them to CNF and then by counting the number of clauses.
- **DomainSize**: number of possible configurations ignoring any kind of constraint of the model.
- **N.valid**: number of valid configurations (considering also the constraint).

All these variables are known to have some impact on the total cost of the test.

B. Data Understanding and Preparation

During this phase, we collect all the data (models and test generation data), we compute the model attributes, and we select the relevant features that can be used as input variables for the prediction model.

Collecting benchmarks: As training instances for CIT problems we have gathered a wide set of 114 models with constraints taken from the literature (Casa [7], [6], [14], FoCuS [15], ACTS [4], and IPO-S [16]) and from SPLOT SPLs repository, and used (in subsets) also by many other papers. The benchmarks can be found on the CitLab web

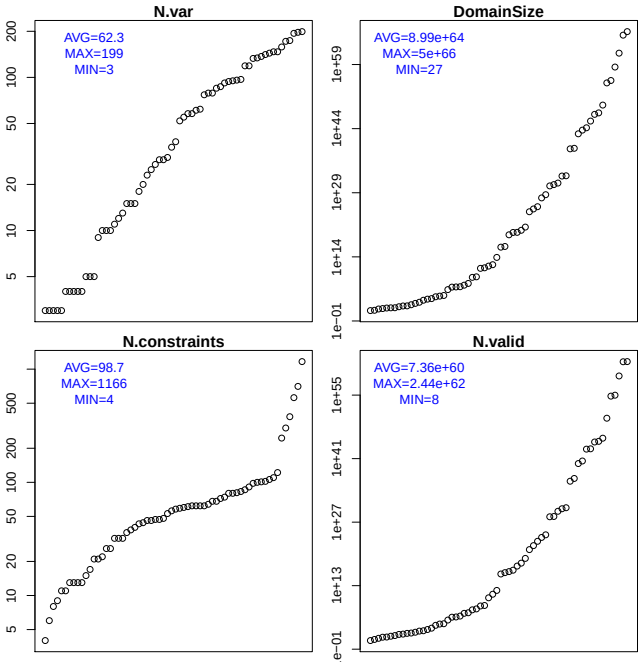

Figure 2: Training set attributes and characteristics

site and they can be used for further comparisons. Fig. 2 shows the distribution of the characteristics we are interested and that could influence the test generation process for the models under observation. The data in Fig. 2 prove that our benchmarks cover a rather wide range for every model attribute.

Test generation data: We performed 50 runs over these 114 models for 4 different generators: all the 3 tools with two configurations of MEDICI. MEDICI, as many other tools, can be fine tuned by using options: we use a fast variant (MEDICI_1_1_1) and a more slow one (MEDICI_10_30_5) which should produce fewer tests [8]. We use the same tool with two different configurations in order to prove that our approach could be used to decide the parameters for a single given tool.

Using an R script we have calculated the total cost for each couple, generator-model, varying the cost of a single test execution UEC in the set {0.01, 0.1, 1, 10, 50, 100, 500, 1000, 5000} seconds.

Model features: In order to gather the data about the models, we use the CITLAB APIs that allow querying the model and obtain structural data like number of variables and so on. Obtaining the number of valid configurations, however, is more complex: one could easily enumerate all the configurations and count those valid. This is very time consuming. Instead, we use the capability of Multi-Valued Decision Diagrams embedded in MEDICI of counting the number of valid paths in a very efficient way.

Correlation features selection (CFS): We select the features that will constitute the inputs of the decision trees. Good candidates should be uncorrelated with each other but highly

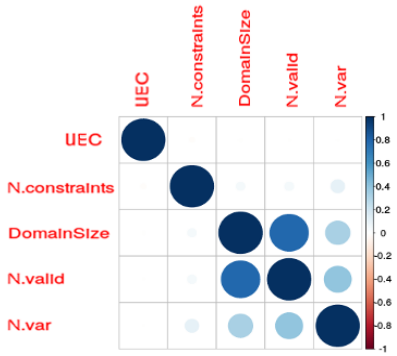

Figure 3: Pearson chart

correlated with the prediction outcome [17]. In our case,the generator portfolio represents the prediction class. We start our feature selection process by the observation of the correlation test of Pearson. Its results are shown in Fig. 3. Pearson's correlation coefficient between two variables is defined as the covariance of the two variables divided by the product of their standard deviations as described by the eq. 2:

$$\rho = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{2}$$

Fig. 3 shows that **UEC** is uncorrelated to other variables (as expected) and it must be considered. Regarding the model features, we note that **N.var** is correlated to all the other variables under test so it is the first candidate to be excluded from our subset of features. Analyzing the Pearson chart, we can also notice that **N.valid** and **DomainSize** are strongly correlated with each other so one of them is the next candidate for the exclusion. Performing an accurate analysis of correlation between the two candidates, we can estimate that they are interchangeable in terms of correlation with the predictive class. This condition led us to exclude **N.valid** because it requires a good amount of time to be computed. Our features subset is composed by: **DomainSize**, **N.Constraints**, and **UEC**.

C. Modeling

In this phase, we use Rattle to produce two predictive models based on decision trees. Exploring some data mining and machine learning techniques we have obtained two different automatic decision makers:

1) **CBT** cost-based decision tree: this classifier is able to select "the best" generator evaluating **only** the test execution cost (**UEC**), ignoring all the features of the model under test.
2) **FBT** feature-based decision tree: this classifier receives as inputs also the domain size of the model, numbers of constraints and the cost for a single test execution and predicts the best generator.

Building the training/validate/test datasets: Using the functionalities of Rattle we have divided the dataset into three different subsets: *training, validation, and test*. The

training subset is used during the modeling process as base of knowledge for our predictive model (the decision tree), the other two sets are used during the evaluation process.

The R script produced by Rattle and used to build the three subsets follows.

```
# Randomly allocate 70% of the dataset to training,
# 15% to validation, and the remaining 15%
# to testing
set.seed(crv$seed)
crs$nobs <- nrow(crs$dataset)
# Splitting the dataset in Training set (70%)
crs$sample <- crs$train
      <- sample(nrow(crs$dataset), 0.7*crs$nobs)
# Validation set (15%)
crs$validate <- sample(setdiff(seq_len(nrow(crs$
    dataset)), crs$train), 0.15*crs$nobs)
# Test Set (15%)
crs$test <- setdiff(setdiff(seq_len(nrow(crs$dataset
    )), crs$train), crs$validate)
```

Variable selection: After the sub-setting process, we have selected the input features of the decision according to the CFS previously performed, and we have set the total cost (**TC**) as the risk variable and the generator classes as target. The decision tree is in fact used as classifier for the selection of an optimal generator that minimizes the **TC** of testing. In the following paragraphs, we describe the modeling process used to produce **FBT**. **CBT** and **FBT** belong to the same modeling process but **CBT** takes as input variables only **UEC**.

The R script used to select the input variables and the target follows.

```
# The following variable selections have been noted.
# Selecting the three numeric input variable
crs$input <- c("N.constraints", "DomainSize", "UEC")
crs$numeric<-c("N.constraints", "DomainSize", "UEC")
crs$categoric <- NULL
#Imposing the generator classes as target
crs$target  <- "generator"
crs$risk    <- "TC"
crs$ident   <- NULL
# Ignoring the other features
# of the base of knwoledge
crs$ignore  <- c("model", "N.var", "N.valid", "run",
     "size", "time")
crs$weights <- NULL
```

Modeling: We have obtained a compact decision tree using Rattle. Its decision tree functionalities are based on the library **rpart**. Rattle allows the user to set 4 configuration parameters:

- *Min split = argument* specifies the minimum number of observations that must exist at a node in the tree before it is considered for splitting.
- *Max depth = argument* limits the depth of a tree.
- *Min bucket = argument* is the minimum number of observations in any terminal leaf node (conventionally *minsplit=3*minbucket*).
- *Cost complexity (cp) = argument* is used to control the size of the decision tree and to select an optimal tree size. The complexity parameter controls the process of pruning a decision tree.

The following extract of code shows that we required that the minimum number of observations in a node is 50 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.015 (cost complexity factor), we have also limited tree depth to 10. As parameters tuning process performed, we have substantially performed a "backward tuning", where we have obtained the final configuration by generating iteratively a new model by varying one parameter at a time and discarding the model and the change in the configuration parameter if the accuracy of the new model decreased with respect to the previous one. We have pruned back the tree to avoid over-fitting the data. We wanted to select a tree size that minimizes the cross-validated error. Specifically we have iteratively examined the cross-validated error results varying the complexity parameter, and we have selected the *cp* associated with minimum cross-validation error. Rattle uses an information gain measure for deciding between alternative splits. This decision algorithm is based the concept of *Shannon Entropy*. The split that provides the greatest gain in information (and equivalently the greatest reduction in entropy) is the chosen split. The R script used to build the decision tree (FBT) follows.

```
# Building decision tree
# Using the dataSet previously prepared we build
# the decision tree using rpart
crs$rpart <- rpart(generator ~ .,
data=crs$dataset[crs$train,c(crs$input,crs$target)],
 method="class",
 parms=list(split="information"),
    control=rpart.control(minsplit=50,
    minbucket=16, maxdepth=10, cp=0.015000))
```

The two final decision trees CBT and FBT are reported in Fig. 5 and 4. Although FBT is more complex than CBT, both trees are rather understandable and easy to follow in order to get guidance on what tool to use even by hand.

D. Evaluation

During this phase, we evaluate the prediction models obtained in the previous phase. The results of the evaluation are presented in Section IV.

E. Deployment

During this phase we describe how it is possible to reuse our predictive models, computed using Rattle, in a Java application and consequently integrating them in the CITLAB framework. Our deployment is based on the use of the Predictive Model Markup Language (PMML), which is an XML-based file format developed by the Data Mining Group to provide a way for applications to describe and exchange models produced by data mining and machine learning algorithms.

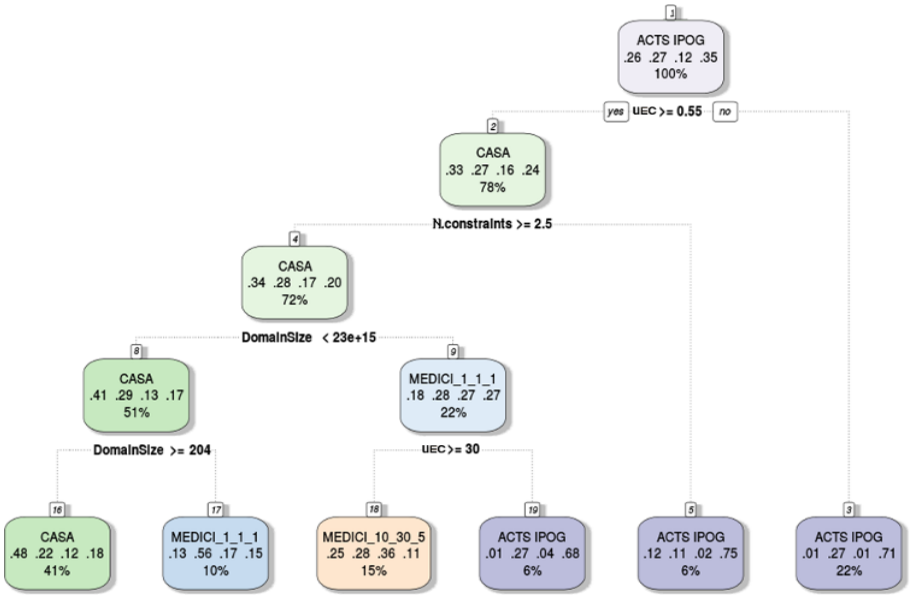

Figure 4: FBT: feature-based decision tree

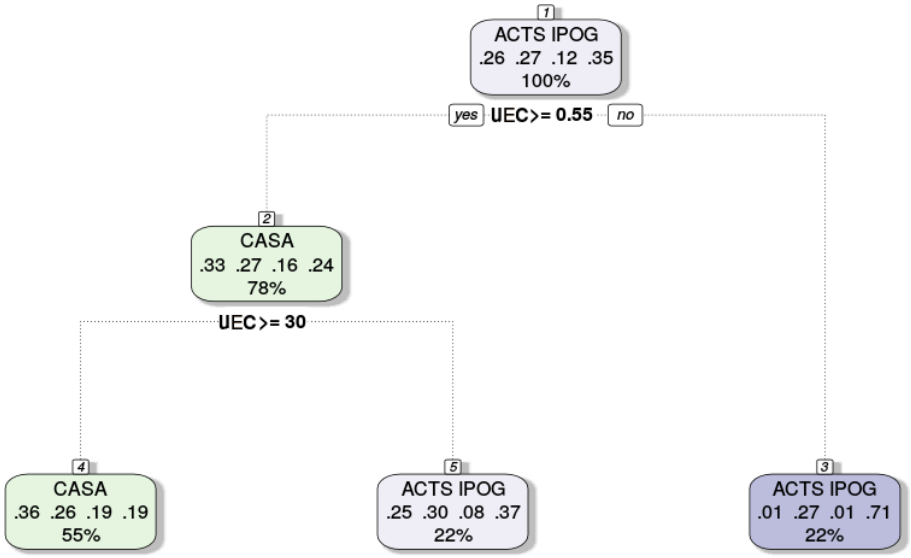

Figure 5: CBT: cost-based decision tree

It supports common models such as logistic regression, feed-forward neural networks and decision trees. Rattle supports the export of predictive models to PMML language. Using JPMML is possible to use PMML models in Java allowing their integration in CITLAB as a plug-in for Eclipse. Using JPMML, we have noticed these pros and cons.

Advantages:

- No lock-in; you can run your models with any library that can read PMML models.
- Runs inside the Java process. No inter-process communication.
- Pure Java solution at run-time.

Disadvantages:

- Supported models depend on the PMML library (for example, JPMML does not support support vector machines SVM, even though it is in the specification of PMML 4.1).

IV. EXPERIMENTS

In the experiments, we want to compare the two decision trees and measure the advantages of using our classifiers against the use of a single generator.

First, we build the best possible predictor one could reasonably have. We identify the generator that gives the minimum average total cost for each model and for each single test cost over all the run we have performed, in order to simulate an *average optimum predictor*. Starting from these data, we build an *optimum* classifier, which would always select on average

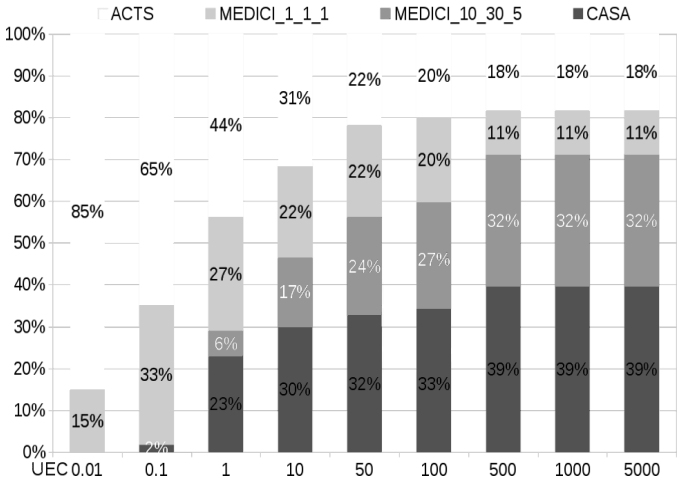

Figure 6: Test generator distribution of the optimum predictor

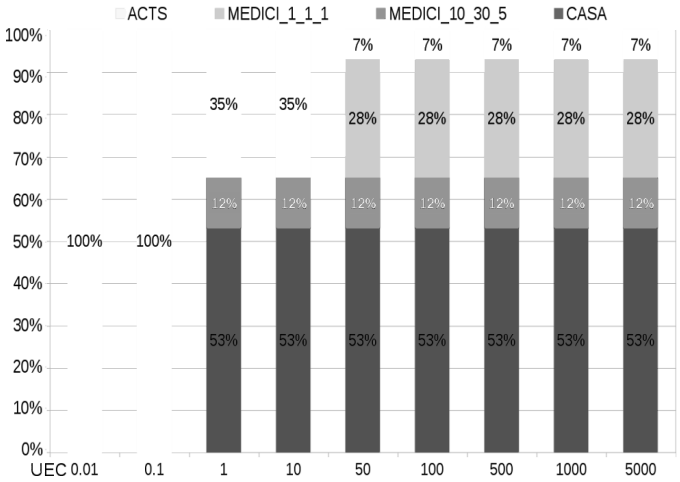

Figure 7: Generation tools distribution using FBT

the best generator for every model. The main problem of the *optimum* predictor is that it can be computed only after the data have been generated ("a-posteriori" predictor).

RQ1 Which generators would choose the optimum predictor?

Fig. 6. shows the distribution of the predicted generators performed by the optimum predictor for each UEC. ACTS is the privileged choice for small test costs due to its fast generation speed while CASA is the best solution for high test costs because it is very slow compared to ACTS but it produces smaller test-suites. MEDICI stays in between ACTS and CASA. MEDICI_1_1_1 produces test suites a little smaller then ACTS's ones but it is 10 times slower, MEDICI_10_30_5 produces test suites comparable to CASA but it can be faster [8]. Overall, ACTS is chosen in the 85% of cases for a single test cost of 0.01s and its percentage continuously decreases until 18% at the test cost of 500s while the percentages of the other generators increase. We can estimate that for a single test cost of about 100s to 500s the 4 generators under test reach a sort of stability point in the distribution between the chosen generators. After this point, the increase of cost does not produce any kind of further changes in the percentages of the selection distribution.

The figure confirms that there is no one *best* generator. By varying the UEC the distribution of generator choices may sensibly change. The data also show that, even the test cost were fixed, the choice of the best generator depends on the model features. This confirms the validity of our assumptions.

RQ2 When does FBT choose a wrong generator?

Fig. 7 shows the distribution of generators selected by FBT. It differs from the optimum reported in Fig. 6. Gray cells of Tab. I display the *confusion matrix* of the predictor. The difference between the optimum and the predicted one is reported as percentage of generator confusion. A confusion matrix displays the percentage of correct or incorrect predictions made by a classifier such as a Bayesian network or decision trees. It is automatically computed by Rattle using the test set. Diagonal elements of the matrix show the percentage of correct predictions, while off-diagonal elements show incorrect predictions. The sum of diagonal values represents the accuracy of the classifier under validation process. False negative rate (FNR) is computed, for each row, as the ratio between the sum of off-diagonal values (false negatives) and the sum of each value of the row values (false negatives + true positives). The high FNRs of the two configurations of MEDICI show a criticality of FBT in their right identification and prediction. FBT presents an accuracy of 58%.

Table I: FBT confusion matrix for the test set

Optimum choice	Predicted				
	CASA	M_1_1_1	M_10_30_5	ACTS	FNR[2]
CASA	**22%**	1%	4%	1%	25%
M_1_1_1	8%	**6%**	4%	7%	76%
M_10_30_5	5%	2%	**5%**	0%	58%
ACTS_IPOG	7%	1%	2%	**24%**	29%
Accuracy[3]			58%		

Table II: CBT confusion matrix for the validation set

Optimum choice	Predicted				
	CASA	M_1_1_1	M_10_30_5	ACTS	FNR
CASA	**22%**	0%	0%	6%	21%
M_1_1_1	13%	**0%**	0%	12%	100%
M_10_30_5	11%	0%	**0%**	2%	100%
ACTS_IPOG	11%	0%	0%	**24%**	31%
Accuracy			46%		

RQ3 When does CBT choose a wrong generator?

[2]False negative rate is the proportion of events that are being tested for which yield negative test outcomes with the test, i.e., the conditional probability of a negative test result given that the event being looked for has taken place. False Negative Rate = (false negative)/(true positive + false negative).

[3]accuracy=(true positives + true negatives)/(positives + negatives)

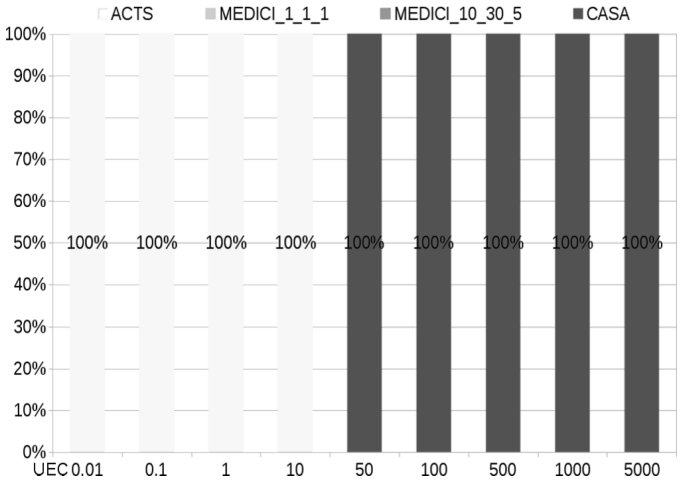

Figure 8: Generation tools distribution using CBT

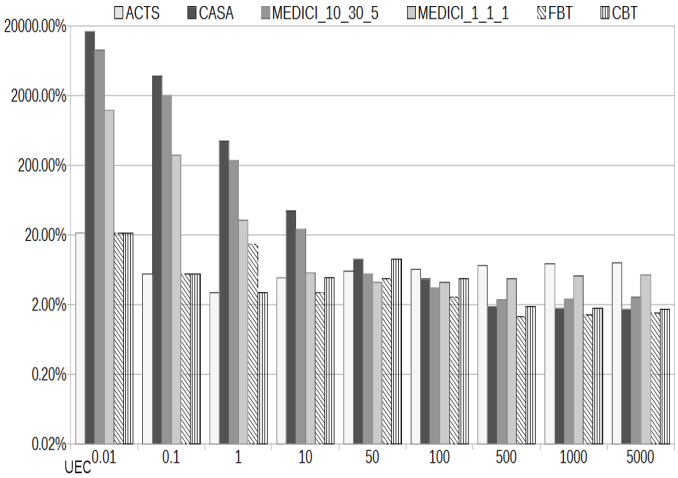

Figure 9: Performance comparison of Predictors versus the use of a fixed tool using the optimum as measure of comparison

Fig. 8 shows a distribution of selected generators that differs, as expected, from the optimum reported in Fig. 6. The difference between the optimum generator and the predicted one is reported in Tab. II as percentage of generator confusion. We notice that CBT performs a trivial choice between ACTS_IPOG and CASA ignoring the other 2 generators. For MEDICI, the FNRs are 100%. CBT decision exchanges MEDICI_10_30_5 to CASA because MEDICI generator performance, in terms of size, is similar to CASA's one but it differs from it in terms of time consumption at the varying of models complexity. The reduction of FBT model to CBT decreases the accuracy rate of 12% (from 58% to 46%). These data suggest that a right estimation of the total cost produced by a generator and a right selection of the generator should consider other feature besides the UEC.

RQ4 Can our predictors outperform the other fixed generators?

Table III reports the total cost over all the combinatorial models for the optimum predictor in seconds and the percentage of variation w.r.t. the optimum for all the other generators by varying the single test cost. The numbers in bold are the minimum values for each cost with the exception of the optimum. It shows that some generators performs very well when the single test cost is small and other ones performs very well when the single test cost increases. It shows that FBT outperforms fixed generator selection in 7 cases over 9. In only one case, the FBT is defeated by CBT. For an UEC of 50 both the decision trees are defeated by MEDICI_1_1_1 that is never selected by CBT. The gap between both decision trees and the optimum a-posteriori predictor makes the possibility of further improvements, in the prediction strategy, concrete.

RQ5 How much is the gain achieved by using decision trees?

Fig. 9 shows the performances of the two predictors CBT e FBT versus the use of a fixed generation tool. The data are presented as percentage of the difference between the total cost for a single generator or predictor and the total cost obtained by the optimum predictor. The percentage of improvement of the total cost achieved using the decision tree decreases with the augment of the single test cost but in terms of absolute time saving is very advantageous. The difference between the two predictors is rather small, but the difference with a single test generator can be quite high. This proves that choosing a single test generator for every test generation task can lead to a significant loss of time.

RQ6 How much are decision trees better than a random selection?

We have compared decision trees with a random selector that randomly choses the test generators. A random selector has the advantage that it does not need anything to make the prediction. The decision trees strategies always outperform a random selection policy as shown in Fig. 10. The gap of performance decreases with the augment of the single test cost but it remains significant if compared to the results of the optimum.

V. Threats to validity

Our findings are subject to the following threats to validity. First, our decision tree models may fail to predict the right test generator for a particular model and UEC. Indeed, our models can give only a statistical estimation, but they cannot guarantee to find the best solution. However, the wide range of models we have used to build the decision trees and the extensive set of UECs can give a good confidence that the right generator is most likely chosen and even if this is not the case, the loss of time is always rather small. Even CBT, as shown in Table III, behaves on average of all the models as the best fixed generator for almost each UEC. One of the most significant issues that afflicts decision tree models is the overfitting of training data which, produces a loss of performance

Table III: Total Mean Cost varying single test cost

	Single test cost – UEC (seconds)								
	0.01	0.1	1	10	50	100	500	1000	5000
Optimum	87.79	388.75	3349.11	32268.57	159313.06	317345.41	1573761.56	3136876.52	15635276.52
ACTS	21.14%	5.49%	2.94%	4.82%	5.97%	6.37%	7.23%	7.59%	7.93%
CASA	16770.17%	3783.26%	436.13%	44.26%	8.99%	4.77%	1.88%	1.75%	1.69%
MEDICI_10_30_5	8900.82%	2006.81%	230.68%	23.71%	5.52%	3.47%	2.32%	2.42%	2.54%
MEDICI_1_1_1	1231.65%	277.01%	32.32%	5.65%	**4.14%**	4.21%	4.78%	5.10%	5.40%
FBT	**21.11%**	**5.48%**	14.66%	**2.93%**	4.72%	**2.56%**	**1.33%**	**1.41%**	**1.52%**
CBT	21.11%	5.48%	**2.94%**	4.82%	8.99%	4.77%	1.88%	1.75%	1.69%
Random	6567.49%	1586.49%	201.61%	18.99%	6.25%	4.94%	4.14%	4.22%	4.39%

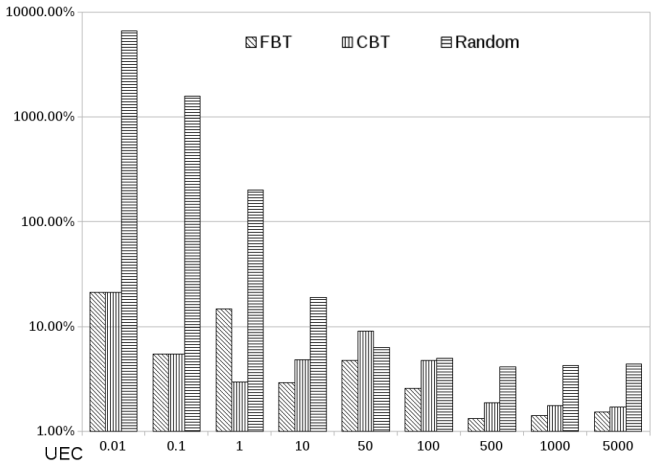

Figure 10: Performance comparison of predictors versus random selection of generators using the optimum as measure of comparison

on new data. In general, when a decision tree model is too complex is unable to correctly match new, previously unseen data. The process that governs the complexity of a model is "Pruning". We have tried to reduce the impact of this issue iteratively pruning our models in order to find the less complex model that presents and acceptable error rate according to *Cost complexity pruning* methodology.

VI. Related Work

Algorithm selection problem is widely studied in literature. John R. Rice formalized the concept of algorithm selection in [18] seeking to answer the question: "Which algorithm is likely to perform best for my problem?". In the early 1990's, the scientific community recognizes Algorithm selection problem as a learning task so, the machine learning community has developed the field of meta-learning, focused on learning about learning algorithm performance on classification problems. M. G. Lagoudakis and M. L. Littman in [19] consider the problem of algorithm selection: dynamically choose an algorithm to attack an instance of a problem with the goal of minimizing the overall execution time. This conception is similar to the one that aimed our approach even if they formulate the problem as a kind of Markov decision process (MDP), and use ideas from reinforcement learning to solve it while we used decision tree technique. In our paper, we focused on the use of machine learning techniques applied to the combinatorial testing algorithm selection in order to minimize the sum of generation time and execution time for the generated test- suites. The SATzilla team in [20] made an interesting observation about the absence of a *dominant* SAT solver and about the fact that different solvers perform best on different instances. They suggest, rather than following the traditional approach, to choose the best solver for a given class of instances. We notice a similar situation for the different generation algorithms for combinatorial interaction testing and we advocated the need to aid the practitioners in algorithm selection. Their approach takes as input a distribution of problem instances and a set of component solvers, and constructs a portfolio optimizing a given objective function (such as mean runtime, percent of instances solved, or score in a competition). We automated the selection of an algorithm at a time while they execute different algorithms in parallel. We plan to implement parallel execution too and to set different execution timeouts for the different algorithms according to the different probabilities given by our classifier. Very few works that apply machine learning techniques to combinatorial testing. Jia at al. presents in [21] an algorithm for combinatorial interaction testing based on a Hyperheuristic search. They have implemented a reinforcement learning agent that is iteratively used for the tuning of the configuration parameters of the simulation annealing operators during the test-suite generation.

VII. Future works

We plan to investigate on the influence of the complexity of constraints over the generation time and over the size of the test-suites. The cost function we have used in this paper, could be improved by considering the complexity of the constraints and the cardinality of the parameters that compose the domains of the models under test. Another factor to consider could be the number of attempts needed to produce the final test-suite. When the system under test is very complex, it is necessary to run the generation tools many times during the different stages of software testing process. We want also to experiment other predictive techniques like *support vector machines* which seem, by a preliminary study, to have a more accuracy in terms of classification but they need further tunings to be deployed in Java.

VIII. Conclusions

The combinatorial testing community has produced many algorithms, techniques, and tools for test generation in these years. Even if one considers only the performances, it remains unclear which is the best solution. For example, a very fast tool may produce a very big test suite that would require much more time during the test execution. In this paper we have introduced a simple cost model for comparing test generation tools and we have shown that there is no best combinatorial tests generator. Depending on the cost of executing a single test, a tool may be more suitable than another. In addition, combinatorial model characteristics should play a role in the choice of the generator. In order to automatize the decision of the right test generator, we have devised a data-mining process able to produce two decision trees to be used in the choice. Experimental results show that our decision trees can efficiently help the tester in the generation process with a significant reduction of the total testing cost. However we can isolate two points that are not worthwhile for our approach adoption. The first weakness of our methodology is that our predictor models take some data as inputs like number of variables, number of constraints, and so on, that may require some time to be computed. The time spent to find model features may be greater than the time saved by choosing the right generator. For this reason, we have chosen only very simple model characteristics and we have devised the CBT model, which requires only UEC. Moreover, we have ignored **N.valid** as input because it is rather costly to be computed. Computing all the model features used by FBT requires around 3.5 seconds for all the models.

The cost of switching to a general framework as CitLab that allows the use of multiple generators may be not worthwhile for researchers using already a specific tool. In addition in this case, our work can be used as guidance to check if the chosen tool is suitable for the testing tasks to be performed. For instance, a serious loss of time is very likely to occur if the tester uses ACTS for generating tests which require a lot of time each to be executed. Our study shows that the random choice can lead to very high time losses.

References

[1] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial interaction testing with CitLab," in *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tool track*, 2013.

[2] A. Gargantini and P. Vavassori, "CitLab: a laboratory for combinatorial interaction testing," in *Workshop on Combinatorial Testing (CT) In conjunction with International Conference on Software Testing (ICST 2012, April 17-21)*, Montreal, Canada, 2012, pp. 559–568.

[3] B. A. Huberman, R. M. Lukose, and T. Hogg, "An economics approach to hard computational problems," *Science*, vol. 275, pp. 51–54, 1997.

[4] "Advanced Combinatorial Testing System (ACTS)." [Online]. Available: http://csrc.nist.gov/groups/SNS/acts/

[5] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, Sep. 2008.

[6] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

[7] "CASA: Covering arrays by simulated annealing." [Online]. Available: http://cse.unl.edu/citportal/tools/casa/

[8] A. Gargantini and P. Vavassori, "Efficient combinatorial test generation based on multivalued decision diagrams," in *Hardware and Software: Verification and Testing, Haifa Verification Conference HVC 2014*, ser. Lecture Notes in Computer Science, E. Yahav, Ed., vol. 8855. Springer International Publishing, 2014, pp. 220–235.

[9] J. Zhang, Z. Zhang, and F. Ma, *Automatic Generation of Combinatorial Test Data*, ser. SpringerBriefs in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, 00000.

[10] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv*, vol. 43, no. 2, p. 11, 2011.

[11] G. Williams, *Data Mining with Rattle and R*, Springer, Ed. NY: Springer New York, 2011.

[12] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[13] C. Shearer, "The crisp-dm model: the new blueprint for data mining," *Journal of data warehousing*, vol. 5, no. 4, pp. 13–22, 2000.

[14] M. Cohen, M. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *Software Engineering, IEEE Trans. on*, vol. 34, no. 5, pp. 633–650, 2008.

[15] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 254–264.

[16] A. Calvagna and A. Gargantini, "T-wise combinatorial interaction test suites construction based on coverage inheritance," *Software Testing, Verification and Reliability*, vol. 22, no. 7, pp. 507–526, 2012.

[17] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.

[18] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[19] M. G. Lagoudakis and M. L. Littman, "Algorithm selection using reinforcement learning." in *ICML*, 2000, pp. 511–518.

[20] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res. (JAIR)*, vol. 32, pp. 565–606, 2008.

[21] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction testing strategies using hyperheuristic search," University College London, Department of Computer Science, Research Note RN/13/17, 2013.