View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Software Testing with Active Learning in a Graph

Nicolas Baskiotis, Michèle Sebag, and Marie-Claude Gaudel

LRI – CNRS – INRIA
Université Paris-Sud, F-91405 Orsay Cedex, France

Abstract. Motivated by Structural Statistical Software Testing (SSST), this paper is interested in sampling the feasible execution paths in the control flow graph of the program being tested. For some complex programs, the fraction of feasible paths becomes tiny, ranging in $[10^{-10}, 10^{-5}]$. When relying on the uniform sampling of the program paths, SSST is thus hindered by the non-Markovian nature of the "feasible path" concept, due to the long-range dependencies between the program nodes. A divide and generate approach relying on an extended Parikh Map representation is proposed to address this limitation; experimental validation on real-world and artificial problems demonstrates gains of orders of magnitude compared to the state of the art.

1 Introduction

The increasing complexity of computer science systems has brought on new demands for systems able to configure, adapt and repair themselves, leading to the emergence of the Autonomic Computing field. As autonomic systems should be "self-aware" (endowed with a behavioural model of themselves), Autonomic Computing is becoming a source of challenging applications in Machine Learning (see e.g., [1, 2]). Similar demands are brought on in the software industry, and ML approaches have been used for software assessment [3], debugging [4] or testing [5].

This paper is interested in structural statistical software testing [6]; the central question is to bridge the gap between the syntax of the program (the control flow graph) and its semantics (the paths in the graph which actually correspond to execution paths, referred to as *feasible* paths). When feasible paths are a tiny fraction $(10^{-15}, 10^{-10})$ of the paths, the application goal is to construct a path sampling mechanism biased toward feasible *new* paths. The difficulty is that by construction very few feasible paths are initially available due to their cost, on the one hand; and on the other hand, the target concept of "feasible path" is non-Markovian (the program semantics involves long-range dependencies among the nodes in the control flow graph). The contribution of the paper concerns active learning in the control flow graph. The failure of baseline probabilistic approaches is observed and explained; a divide and sample mechanism is presented, providing gains of orders of magnitude compared to the state of the art on real-world and artificial problems.

Dagstuhl Seminar Proceedings 08351
Evolutionary Test Generation
http://drops.dagstuhl.de/opus/volltexte/2009/2014

Section 2 introduces the application domain and the position of the problem. Section 3 describes the proposed approach. Section 4 reports on the experimental validation and discusses the approach with respect to related work. The paper concludes with some perspectives for further research.

2 Position of the problem

Software testing comes in two main flavors, the algebraic and the statistical one. The algebraic approach is interested in model checking, and proving that the program satisfies the desired properties (derived from its specifications or provided by the expert). The statistical approach is interested in constructing a set of test cases; on every test case (values of the input variables of the program), one compares the actual output of the program with the desired output. The latter approach allows one to bound the probability of errors in the program along the PAC framework, based on an appropriate distribution of the test cases.

The most natural idea for constructing test cases is to sample uniformly the domain of the input variables. However, it is easy to see that uniform sampling will but miss the exception branches (e.g. calling the division routine with denominator $= 0$), the measure of which is null. More generally, uniformly sampling the input domain does not result in a good coverage of the execution paths of the program (80% of the input domain exert less than 20% of the feasible paths).

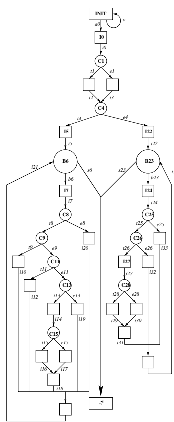


Fig. 1. Control flow graph of Program FCT4 (36 nodes, 46 edges).

To address the above limitation, the statistical structural software testing approach [6] proceeds by uniformly sampling the execution paths of the program being tested after its control flow graph (Fig. 1). The control flow graph is a Finite State Automaton described from some alphabet Σ (conditions and instructions nodes of the program) and the set of transitions between nodes. For every node v, $Suc(v)$ denotes the set of its successor nodes. Since programs being tested generally include loops (`while` or `for` instructions), the path length is not bounded a priori. However for practical reasons, a maximal path length is set, chosen by the test expert. Classical results from labelled combinatorial structures [7] are then used to uniformly sample the set of paths with length T. Finally, each program path is converted into a constraint satisfaction problem (CSP), expressing the constraints which the input variables should satisfy in order to follow this path; if this CSP is satisfiable (equivalently the path is said to be *feasible*), its solution gives the test case exerting the path (Fig. 2).

While structural statistical software testing ensures the uniform sampling of the program behaviors, the drawback is that the fraction of feasible paths might

Program

```
1   read (x, y);
2   if (x < 0)
3       then x := -x;  y := 1/y;
4   p := 1;
5   while (x > 0)
6       do p := p * y; x := x - 1;
7   print p;
```

Path

$s = 1.2.4.5.7$

Path → Constraint Satisfaction Problem

$x \geq 0$ AND $x \leq 0$

CSP → Test case

$x = 0$

Fig. 2. From program paths to test cases

be tiny for programs involving nested loops and complex variable dependencies; in such cases, the test expert can but manually modify the program to render the uniform sampling approach efficient. The low percentage of feasible paths reflects the huge gap between the syntactic description and the semantics of the program. Specifically, a major cause for path unfeasibility, referred to as XOR patterns, is to violate the dependencies between different parts of the program. For instance, if the program involves two `if` nodes based on some (unchanged) expression, the successor nodes of these `if` nodes will be correlated in every feasible path: if the successor of the first `if` node is the `then` (respectively, `else`) node, then the successor of the second `if` node must be the `then` (resp. `else`) node.

3 Overview

After describing the path representation used in the rest of the paper, this section analyzes the failure of baseline probabilistic generative learning for the problem domain. A divide and sample algorithm addressing these limitations is finally presented.

3.1 Extended Parikh Maps

A first phase of the study has been concerned with supervised learning from the training set $\mathcal{E} = \{(s_i, y_i), s_i \in \Sigma^T, y_i \in \{-1, +1\}, i = 1 \ldots, n\}$ made of the few feasible paths ((s_i, y_i) with $y_i = 1$) and many unfeasible paths ((s_i, y_i) with $y_i = -1$) collected by uniform sampling and labelled by the oracle (constraint solver, CS). The goal was to approximate the semantics of the program (more precisely to estimate the "feasible region") to filter out most unfeasible paths, thereby saving many useless calls to the CS routine (a few seconds per call). As could have been expected, standard ML approaches pertaining to Grammatical Inference or Hidden Markov Models failed to learn the "feasible region" concept; indeed the long-range dependencies in this concept make it non Markovian. More sophisticated approaches, e.g. [8], were hindered due to the insufficient amount of feasible paths.

A frugal representation inspired from Parikh Maps [9] was chosen to accommodate the sparsity of the available examples. Parikh maps use the Ngrams built from alphabet Σ as features; every string is described as an integer vector reporting the number of occurrences of every Ngram in the string. Despite their simplicity, Parikh maps finely capture the grammar generating the strings. As shown by [10], the probability from two grammars to be different can be upper bounded from the distance between their N-grams distributions in a string sample (depending on the length N of Ngrams, the sample size and the distance). As shown by [11], mildly context sensitive grammars can be characterized from equations in the Parikh map representation. However, due to the number of available examples (a few dozens) relatively to the complexity of the grammar (a few dozen symbols in the alphabet, a few hundred nodes in every path), standard Parikh maps cannot not be used directly: 1-grams do not provide enough information to enable discriminant learning while 2-grams result in a too sparse representation.

An extended Parikh map representation is thus proposed. Additional attributes specify for each pair $(v, i) \in \Sigma \times \mathbb{N}$ the successor node of the i-th occurrence of the v symbol in the path. Formally, the extended Parikh map representation involves $|\Sigma| \times I$ attributes, where I is an upper bound on the number of occurrences of every symbol in any path s:

$$\begin{array}{lll} v \in \Sigma & |\cdot|_v : \Sigma^* \mapsto \mathbb{N} & |s|_v = \#v \; in \; s \\ (v, i) \in \Sigma \times \mathbb{N} & |\cdot|_{v,i} : \Sigma^* \mapsto \Sigma & |s|_{v,i} = \text{ successor of i-th occurrence of } v \end{array}$$

Standard supervised learning approaches (SVM, decision trees) using the extended Parikh map representation still failed to learn the "feasible path region"; this failure was blamed on the insufficient number of feasible paths in the training set. At this point it became clear that not only would the acquisition of additional feasible paths enable supervised learning; also, it would directly address the limitations of structural statistical software learning problem.

3.2 The Failure of Probabilistic Active Learning

The second phase of the study thus focuses on the construction of additional new feasible paths. Taking inspiration from [8], let the current path s be initialized to the starting node v_s. At step t, the point is to select the successor of the current symbol $s[t]$ in order to maximize the probability of arriving at a new feasible path:

$$s[t+1] = argmax_{w \in Suc(s[t])} Pr(s' \text{ feasible and new } | \text{ Prefix}(s') = sw) \quad (1)$$

Since there is no feasible path s' in the training set with prefix sw after the very first steps ($t \geq 5$), a generalized formulation is considered. Let v be the last symbol in s ($s[t] = v$) and assume that s currently involves i occurrences of v ($|s|_v = i$). Then condition Prefix$(s') = sw$ is generalized into: the successor of the i-th occurrence of v is w and s' includes strictly more occurrences of symbol

w than s. Accordingly the selection criterion becomes:

$$s[t+1] = argmax_{w \in Suc(s[t])} Pr(s' \text{ feasible and new } \mid (|s'|_{v,i} = w) \wedge (|s'|_w > |s|_w)) \quad (2)$$

Although it improves on uniform path sampling, the above generative heuristics is still found to be poorly efficient. This failure is blamed on the highly disjunctive structure of the underlying target concept. Specifically, the feasible path region h^* involves the conjunction of many XOR concepts (section 2); h^* is thus expressed as the disjunction of many conjunctive concepts C_i in the Parikh representation. In eq. (2), all feasible paths s' are considered, mixing the contributions from different C_is and thereby misleading the selection criterion[1], as illustrated on the toy problem in Fig. 3. Let h^* be defined as: the first and third occurrences of symbol v have same successor symbol.

	$\lvert \cdot \rvert_{v,1}$	$\lvert \cdot \rvert_{v,2}$	$\lvert \cdot \rvert_{v,1}$	y
s_1	w	w'	w	1
s_2	w'	w	w'	1
s_3	w	w	w'	-1

Fig. 3. Training set $\mathcal{E}$

The training set includes one feasible path in each conjunctive concept of h^* and the unfeasible path s_3. Let the current string be $s = vw'vwv$; eq. (2) leads to select w' as next symbol since

$$Pr(s' \ feasible \mid \ |s'|_{v,3} = w' \wedge |s'|_{w'} > 2) = 1/2$$
$$Pr(s' \ feasible \mid \ |s'|_{v,3} = w \wedge |s'|_w > 1) = 1$$

In the considered context, probabilistic active learning thus raises the following dilemma: If conditional probabilities are based on too specific conditions they are useless; if they are based on too general conditions, they are misleading as the contributions from different conjunctive subconcepts are mixed. A two-step process is proposed to address this limitation, firstly identifying the conjunctive subconcepts represented in the training set (section 3.3) and thereafter stochastically generating new paths to generalize these subconcepts (section 3.4).

3.3 Divide to conquer

In the rest of the paper, "feasible paths" and "positive examples" are used interchangeably.

Definition: $\mathcal{R}(s, s')$
For a given representation language $\mathcal{L}$, let the target concept be expressed as the disjunction of conjunctive concepts C_i, $i = 1 \ldots K$. Let s and s' be two positive examples; define $\mathcal{R}(s, s')$ to be true iff there exists C_j covering both s and s'.

Proposition: *Approximating* $\mathcal{R}(s, s')$
Let $s''_1, \ldots, s''_M$ be M paths uniformly and independently sampled in the least

[1] Similar difficulties are encountered when learning XOR concepts with decision trees.

general generalization[2] of s and s'. Let $\widehat{\mathcal{R}}(s,s')$ be set to $false$ iff at least one among the s''_j is negative (unfeasible), and $true$ otherwise. Then:

$$Pr(\widehat{\mathcal{R}}(s,s')\ false \mid \mathcal{R}(s,s')\ true) = 0 \quad (3)$$

$$q = Pr(\widehat{\mathcal{R}}(s,s')\ true \mid \mathcal{R}(s,s')\ false) \text{ goes to 0 exponentially fast with } M \quad (4)$$

Proof.
(3) follows from the logical structure on $\mathcal{L}$. If $\mathcal{R}(s,s')$ holds then by definition $lgg(s,s')$ is generalized by some C_j, which implies that every example in $lgg(s,s')$ is positive.
(4) follows from $q = p^M$ where $p = Pr(s''\ positive \mid s'' \in lgg(s,s'), \mathcal{R}(s,s')\ false)$.

The conjunctive subconcepts represented in the training set are identified from the cliques defined by relation $\widehat{\mathcal{R}}$, using the Divide algorithm (Fig. 4).

Algorithm Divide
Input: feasible path s
Initialize $\widehat{C}_s = \{s\}$
Define $H_s = \{s' \notin \widehat{C}_s \ / \ \forall\ s'' \in \widehat{C}_s,\ \widehat{\mathcal{R}}(s',s'')\}$
Define $Degree(s') = |\{s'' \in H_s \ / \ \widehat{\mathcal{R}}(s',s'')\}|$
While $H_s \neq \emptyset$
    Select $s' \in argmax_{s'' \in H_s}\{Degree(s'')\}$
    $\widehat{C}_s := \widehat{C}_s \bigcup \{s'\}$
    $H_s := H_s \setminus \{s'\}$
Return $\widehat{C}_s$

Fig. 4. Algorithm Divide

For every feasible path s, let $\widehat{C}_s$ be initialized to $\{s\}$. Let H_s be defined as the set of all examples s' that do not belong to $\widehat{C}_s$ and are linked to every example in $\widehat{C}_s$:

$$H_s = \{s' \notin \widehat{C}_s \ / \ \forall\ s'' \in \widehat{C}_s,\ \widehat{\mathcal{R}}(s',s'')\}$$

Let $degree(s')$ be defined as the number of elements s'' in H_s such that $\widehat{\mathcal{R}}(s',s'')$. While H_s is not empty, iteratively select s' with maximal degree in H_s, add it to $\widehat{C}_s$ and update H_s accordingly.

Proposition: *Identifying conjunctive subconcepts represented in the training set.*
Let s be a positive example, belonging to a single conjunctive concept C and assume that C has $m > 1$ representatives in the training set. Let $\widehat{C}_s$ be constructed by the Divide algorithm.
Let $P_{Err,i}$ denote the probability of selecting at step $i > 1$ an example which

[2] The least general generalization of two examples is defined as the (unique) most specific conjunction covering both examples in the representation language.

does not belong to C, conditionally to the fact that all previously selected examples belong to C. Then $E[P_{Err,i}]$ decreases exponentially with $m-i$ and linearly with q^i, where $q = Pr(\widehat{\mathcal{R}}(s,s')\ true \mid \mathcal{R}(s,s')\ false)$.

Proof.
At the first step, H_s includes all m representatives of C except s itself. With probability q^r, H_s also includes r examples not belonging to C, referred to as spurious examples.
The degree of a spurious example depends on i) its number of links to the representatives of C, which follows the Bernoulli law $\mathcal{B}(m-1,q)$; ii) its number of links to the other spurious examples, upper bounded by $r-1$. Symmetrically, the degree of a non-spurious element is lower-bounded by $m-1$. The probability for a spurious example in H_s to be selected in step 2 of the Divide algorithm is thus upper bounded as:

$$\begin{aligned} A(r) = Pr(\mathcal{B}(m-1,q) > m-r) &= Pr(\mathcal{B}(m-1,1-q) < r-1) \\ &\leq exp\left(-\tfrac{2}{m-1}\left((m-1)(1-q)-(r-1)\right)^2\right) \end{aligned}$$

Summing over all r spurious examples and taking the expectation over $r=1$ to $n-m$, it comes:

$$\begin{aligned} E[P_{Err,1}] &= \textstyle\sum_{r=1}^{n-m} r\, q^r\, A(r) \\ &\leq B \textstyle\sum_{r=1}^{n-m} r \times (C)^{r-1} \times D^{(r-1)^2} \\ with \quad & B = exp\left(-2(m-1)(1-q)^2\right) q; \quad C = q\, exp(4(1-q)); \quad D = exp(-\tfrac{2}{m-1}) \end{aligned}$$

It comes

$$E[P_{Err,1}] < B\, \tfrac{1}{(1-qC)^2}\, \tfrac{1}{(1-D)^2} = \mathcal{O}(F^{-(m-1)} \times q)$$

Therefore $E[P_{Err,1}]$ decreases exponentially with $m-1$ and linearly with q. The same analysis shows that the expectation of selecting a first spurious element at the i-th step of the Divide algorithm decreases exponentially with $m-i$ and linearly with q^i.

The above result[3] shows that, provided that a conjunctive subconcept C has at least one representative in the training set $\mathcal{E}$, one can identify with high probability the subset $\widehat{C} = C \cap \mathcal{E}$. The next step is to identify C from $\widehat{C}$.

3.4 Generalizing subconcepts

Every $\widehat{C}$ constructed by the Divide algorithm is considered independently. The identification of the underlying conjunctive concept C is done iteratively by generating paths which are close to $\widehat{C}$, do not belong to the lgg of $\widehat{C}$, and are feasible. Several approaches have been investigated.

---

[3] The sampling of $lgg(s,s')$ used to define $\widehat{\mathcal{R}}(s,s')$ is done using eq. (2). The limitations described in section 3.2 are avoided since $lgg(s,s')$ is conjunctive by construction.

The first approach is based on a multi-armed bandit formalization [12]; every successor node is viewed as a bandit arm; the associated reward probability is the probability of ultimately getting a feasible path. More precisely, the construction of the current path, involving a sequence of node selections, is formalized as a tree-structured multi-armed bandit and tackled using the UCT algorithm [13]. However UCT encounters two difficulties. On one hand, structural statistical software testing defines a dynamic multi-armed bandit, in the sense that the reward associated to a feasible path becomes null after this path has been found (the goal is to find *new* feasible paths). On the other hand, in the considered problem range UCT fails to construct feasible paths in reasonable time. Its failure is blamed on the very low reward probability relatively to the number of options: the problem boils down to finding the proverbial needle (a feasible path) in the haystack (a sequence of a few hundred choices). The Exploration vs Exploitation policy in UCT requires one to explore the many options which have never been tried, with little hope of finding feasible paths when doing so.

Similar difficulties were reported when adapting UCT in order to build the computer-Go program MoGo [14]. Taking inspiration from MoGo, an ε-greedy strategy is used; the probability of selecting an arm which has never been selected before is ε instead of 1. While the ε-greedy strategy improves on the baseline UCT in the considered context, the bias toward exploration is still too high, no matter what the value of ε is. In most cases, selecting a successor node which had never been selected before engages the path in a new region of the search space and one never gets back to exploitation.

A third approach, called Stochastic Combinatorial Sampling (*SCS*) and inspired from [15], uses two stochastic combinatorial operators to construct new paths from existing paths. Operator μ is a unary operator; $\mu(s)$ is obtained by deleting or inserting in s an admissible fragment, i.e. a subsequence $s[i \dots j]$ such that $s[i-1] = s[j]$. In case of insertion, the fragment is added after a (uniformly selected) occurrence of symbol $s[i-1]$. Operator χ is a binary operator; $\chi(s, s')$ is defined by concatenation of the head of s and the tail of s'. More precisely, $\chi(s, s') = s[1 \dots i].s'[j \dots |s'|]$ where indices i and j are such that $s[i] = s'[j-1]$. In both cases, the resulting path s'' is rejected if i) its length is greater than the maximal path length T; ii) s'' belongs to the lgg of $\widehat{C}$; iii) lgg($\widehat{C} \bigcup s''$) generalizes a negative example (unfeasible path). Every new path s'' is assessed as follows. To every constraint [$attribute = value$] in the lgg of $\widehat{C}$ is associated a weight initially set to 1. The score of s'' is the sum over all conditions in the lgg of $\widehat{C}$ that are violated by s'', of the condition weight.
In every time step, the *SCS* module generates L paths, and sends the path s with minimal score to the oracle to be labelled; if s is feasible, $\widehat{C}$ is updated. Otherwise: i) if s violates a single constraint in $lgg(\widehat{C})$, the associated weight is set to a maximal value (the constraint is a discriminant one); ii) otherwise, the weight of every violated condition is increased.

4 Experimental Validation

This section reports on the experimental validation of the presented approach.

4.1 Experimental Setting

Due to the fact that the oracle (constraint solver, section 2) is a proprietary tool, only four real-world programs taken from [6] could be considered. Three out of four programs could be handled simply using the extended Parikh Map representation (i.e. the semantic constraints were made straightforward to express in this representation). The fourth program (the Fct4 program, involved in the safety of a nuclear plant, Fig. 1), includes 36 nodes and 46 edges; the ratio of feasible paths is circa 10^{-5} for a maximum path length $T = 250$.

A stochastic program generator was thus designed to enable extensive validation (available from the authors). This generator involves a program syntax generator module, using a probabilistic BNF grammar to generate a control flow graph[4], and a program semantics generator module. The latter module defines the target concept h^*, determining whether a given path in the above graph is feasible. After section 2, h^* is a conjunction of XOR concepts. In order to generate satisfiable target concepts h^*, a set $\mathcal{P}$ of paths uniformly generated from the control flow graph is first constructed; iteratively, i) one selects a XOR concept covering a strict subset of $\mathcal{P}$; ii) paths not covered by the XOR concept are removed from $\mathcal{P}$. Finally, the target concept h^* is made of the conjunction of the selected XOR concepts and the lgg of the paths in $\mathcal{P}$. Ten artificial problems are considered with an alphabet size ranging in $[20, 40]$. For a path length in $[120, 250]$ the ratio of infeasible paths ranges in $[10^{-15}, 10^{-3}]$. For each problem, ten independent runs are launched, considering a training set with 50 feasible and 50 infeasible paths[5].

For each conjunctive concept C in h^* which is represented in the training set, an independent set of $100,000$ paths uniformly generated in C is built. It is used to measure the algorithm performance w.r.t. C, by comparing the fraction $i(C)$ of paths in C that belong to the training set, and the fraction $f(C)$ of paths in C constructed after running 200 times the ε-greedy or the *SCS* modules. For the sake of readability, the agregated performance $f(x)$ is computed using a Gaussian convolution over all C represented in the training set (with parameter

[4] Three non-terminal nodes were considered (the generic structure B, the *if* and the *while* structures), together with two terminal nodes (the *Instruction* and the *Condition* nodes). The probabilities on the production rules control the length and depth of the control flow graph. Eventually, the instructions are pruned in such a way that each instruction has at least two successor instructions; finally, each instruction and condition is associated a distinct label.

[5] Increasing the number of infeasible training paths does not make any difference, as only infeasible paths "close" to the feasible ones are relevant, e.g. to define relation $\widehat{\mathcal{R}}$.

$\kappa = 10^-2$):

$$f(x) = \frac{\sum_{C\ s.t.\ i(C)\neq 0} f(C) exp(-\frac{(x-i(C))^2}{\kappa}}{\sum_{C\ s.t.\ i(C)\neq 0} exp(-\frac{(x-i(C))^2}{\kappa}}$$

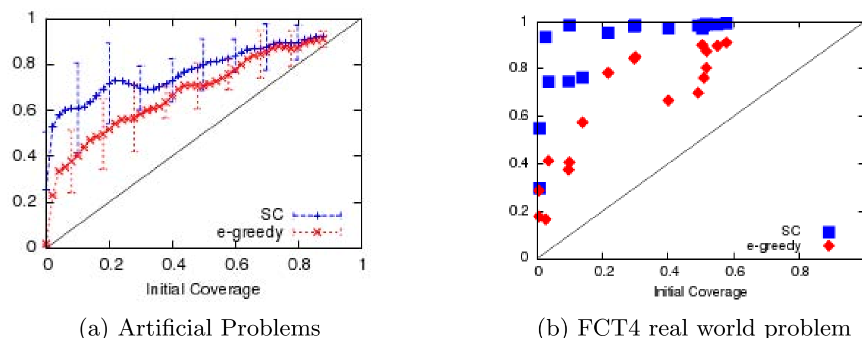
4.2 Experimental results

Fig. 5.(a) displays the performance of the ε-greedy and *SCS* generative approaches on artificial problems. Fig. 5.(b) reports the performances on the real-world fct4 problem comparatively to the state of the art with same experimental setting (3,000 paths generated per run, average results on 10 runs) [5].

Detailed results are reported in Table 1; when the initial coverage of the conjunct is tiny to small (below 10%), the gain ranges from 8 to 2 *orders of magnitude*. A factor gain of 3 is observed when the initial coverage is between 10% and 30%. For concepts which are already well represented in the initial training set, the gain can only be moderate.

These results improve by a few orders of magnitude on the state of the art [6, 5]. The improvement is mostly interpreted as the result of the Divide algorithm, virtually splitting the underlying target concept into simpler, conjunctive, sub-concepts. This interpretation is backed by lesion studies (results omitted due to lack of space) considering ε-greedy and *SCS* generation without the Divide step. The importance of the Divide step should not be underestimated; in essence it turns a non-Markovian decision problem (the choice of a successor node at some point depends on the choices made much earlier) into a set of Markovian ones (at every choice point, the local information can reliably guide the decision).

In terms of computational effort, the runtime of the Divide algorithm ranges between 5 and 15 minutes on PC-Pentium IV; the ε-greedy or *SCS* generative modules require less than 5 minutes per conjunctive concept.



(a) Artificial Problems (b) FCT4 real world problem

Fig. 5. Final vs Initial coverage of the conjunctive subconcepts with ε-greedy and *SCS* generative approaches (averaged on 10 runs, with standard deviation).

		$[0, 10^{-4}]$	$[10^{-4}, 10^{-3}]$	$[10^{-3}, 10^{-2}]$	$[10^{-2}, 10^{-1}]$	$[.1, .3]$	$[.3, .6]$	$[.6, 1]$
$\epsilon - greedy$	$log(f/i)$	5.7 ± 1.2	5.3 ± 1.2	$3.7 \pm .86$	$2 \pm .72$			
	f/i					$3 \pm .1$	$1.6 \pm .3$	$1.1 \pm .1$
SCS	$log(f/i)$	8.2 ± 0.7	7.0 ± 1.4	$5.0 \pm .8$	$2.6 \pm .6$			
	f/i					4.1 ± 1.4	$1.8 \pm .3$	$1.1 \pm .1$

Table 1. Gain obtained with ϵ-greedy generalization for various ranges of the initial coverage of the conjunct.

5 Discussion and Perspectives

The Software Testing approach presented in this paper differs from the Software Debugging approach [4] in two main ways. [4] uses an intrusive approach (inserting instructions, e.g. checking memory, in the program); the execution traces are exploited to identify the most informative inserted instructions (feature selection), which in turn provide hints into the possible causes of the bugs. Compared to Software Debugging, Software Testing intervenes later in the lifecycle of the program; modifying the program is not considered to be appropriate. Further, software debugging observes diversified program behaviours while software testing precisely aims at emulating such diversified behaviours.

A more related work is [5], which likewise addresses Structural Statistical Software Testing. The main difference of the presented approach compared to [5] is the Divide algorithm, which is considered instrumental in the very significant performance gains reported (Fig. 5.(b)). A second difference concerns the use of the *SCS* generative approach. The latter approach is inspired by [15], interested in the quasi-uniform sampling of the solutions of a Constraint Satisfaction Problem. Likewise, [15] exploits some CSP solutions provided by Walksat, and samples the neighborhood of these solutions using a Simulated Annealing algorithm; however, while [15] considers solutions made of boolean vectors, the presented approach deals with the sampling of paths in a graph.

Further research will study the distribution of the presented sampling mechanisms, and investigate the characterization of conjunctive concepts which are not represented in the training set. The limitations of active learning when dealing with a very imbalanced concept in a structured search space will be analyzed and compared to the continuous case [16].

References

1. M. Brodie, I. Rish, and S. Ma. Intelligent probing: A cost-effective approach to fault diagnosis in computer networks. *IBM Systems Journal*, 41(3):372–385, 2002.
2. N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff. Mining for misconfigured machines in grid systems. In *KDD '06: Proc of the 12th ACM SIGKDD International Confe rence on Knowledge discovery and data mining*, pages 687–692. ACM Press, 2006.

3. G. Xiao, F. Southey, R. C. Holte, and D. F. Wilkinson. Software testing by active learning for commercial games. In *AAAI*, pages 898–903, 2005.
4. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML*, pages 1105–1112, 2006.
5. N. Bastiokis, M. Sebag, M.-C. Gaudel, and S.-D. Gouraud. Software testing: A machine learning approach. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence*, pages 2274–2279, 2007.
6. A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE*, pages 25–34, 2004.
7. P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.
8. R. Begleiter, R. El-Yaniv, and G. Yona. On prediction using variable order markov models. *JAIR*, 22:385–421, 2004.
9. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
10. E. Fischer, F. Magniez, and M. de Rougemont. Approximate satisfiability and equivalence. In *Proceedings of 21st IEEE Symposium on Logic in Computer Science*, pages 421–430, 2006.
11. A. Clark, C. C. Florencio, and C. Watkins. Languages as hyperplanes: Grammatical inference with string kernels. In *Proc. Eur. Conf. on Machine Learning*, pages 90–101, 2006.
12. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.
13. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.
14. S. Gelly and D. Silver. Combining online and offline knowledge in uct. In *ICML*, pages 273–280, 2007.
15. Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, pages 670–676, 2004.
16. S. Dasgupta. Coarse sample complexity bounds for active learning. In *NIPS*, pages 235–242, 2005.