A Machine Learning approach for Statistical Software Testing

Nicolas Baskiotis, Michèle Sebag, Marie-Claude Gaudel, Sandrine-Dominique Gouraud

A Machine Learning approach for Statistical Software Testing *

Nicolas Baskiotis, Michèle Sebag, Marie-Claude Gaudel, Sandrine Gouraud
LRI, Université de Paris-Sud, CNRS UMR 8623
Bât.490, 91405 Orsay Cedex (France)
{nbaskiot, sebag, mcg, gouraud}@lri.fr

Abstract

Some Statistical Software Testing approaches rely on sampling the feasible paths in the control flow graph of the program; the difficulty comes from the tiny ratio of feasible paths. This paper presents an adaptive sampling mechanism called *EXIST* for *Exploration/eXploitation Inference for Software Testing*, able to retrieve distinct feasible paths with high probability. *EXIST* proceeds by alternatively exploiting and updating a distribution on the set of program paths. An original representation of paths, accommodating long-range dependencies and data sparsity and based on extended Parikh maps, is proposed. Experimental validation on real-world and artificial problems demonstrates dramatic improvements compared to the state of the art.

1 Introduction

Computer Science is becoming a new application domain for Machine Learning (ML), motivated by the increasing complexity of current systems [Rish *et al.*, 2006]. Ideally, systems should be able to automatically adapt, maintain and repair themselves; a first step to this end is to build self-aware systems, using ML to automatically model the system behaviour. Along these lines, various ML approaches have been proposed for Software Testing [Bréhélin *et al.*, 2001], Software Modeling [Xiao *et al.*, 2005] and Software Debugging [Zheng *et al.*, 2006].

In this paper, we revisit a Statistical Software Testing (SST) approach presented in [Denise *et al.*, 2004]. This approach is based on the uniform sampling of the paths in the control flow graph of the program; to each path, a test case exerting this path can be associated *if* the path is feasible. A problem with this approach is that the control flow graph provides an overly general description of the program; in some cases, and very often for large programs, the fraction of feasible paths is tiny (ranging in 10^{-15}, 10^{-5}).

After a discriminant learning approach failed to characterise the set of feasible paths because of the tiny support of the target concept, a generative learning approach was proposed. This approach, called *EXIST* for *Exploration - eXploitation Inference for Software Testing*, is inspired by both Estimation of Distribution Algorithms (EDAs) [Baluja and Davies, 1998] and Online Learning [Auer *et al.*, 2002; Cesa-Bianchi and Lugosi, 2006]. *EXIST* proceeds by iteratively generating candidate paths based on the current distribution on the program paths, and updating this distribution after the path has been labelled as feasible or infeasible. *EXIST* was made possible by the use of an original representation, extending the Parikh map [Hopcroft and Ullman, 1979] and providing a powerful propositional description of long structured sequences (program paths). Another original contribution, compared to on-line learning [Cesa-Bianchi and Lugosi, 2006; Kocsis and Szepesvári, 2006] or reinforcement learning, is that our goal is to *maximise the number of* distinct *feasible paths* found along the process, as opposed to learning a concept or a fixed policy.

The paper is organised as follows[1]. Section 2 briefly reviews some work relevant to Machine Learning and Software Testing. Section 3 introduces the formal background and prior knowledge related to the SST problem, and describes the extended Parikh representation. Section 4 gives an overview of the *EXIST* system. Section 5 describes the experimental setting and goals, and reports on the empirical validation of the approach on real-world and artificial problems. The paper concludes with some perspectives for further research.

2 Related Work

Interestingly, while Program Synthesis is among the grand goals of Machine Learning, the application of Machine Learning to Software Testing (ST) has seldom been considered in the literature.

Ernst et al. [1999] aim at detecting program invariants, through instrumenting the program at hand and searching for predetermined regularities (e.g. value ranges) in the traces.

Brehelin et al. [2001] consider a deterministic test procedure, generating sequences of inputs for a PLA device. An HMM is trained from these sequences and further used to generate new sequences, increasing the test coverage.

*The first two authors gratefully acknowledge the support of Pascal Network of Excellence IST-2002-506 778.

[1] Due to space limitations, the interested reader is referred to [Baskiotis *et al.*, 2006] for a more comprehensive presentation.

In [Vardhan *et al.*, 2004], the goal is to test a concurrent asynchronous program against user-supplied constraints (model checking). Grammatical Inference is used to characterise the paths relevant to the constraint checking.

Xiao et al. [2005] aim at testing a game player, e.g. discovering the regions where the game is too easy/too difficult; they use active learning and rule learning to construct a model of the program. A more remotely related work presented by [Zheng *et al.*, 2006], is actually concerned with software debugging and the identification of trace predicates related to the program misbehaviours.

In [Ernst *et al.*, 1999; Vardhan *et al.*, 2004], ML is used to provide better input to ST approaches; in [Bréhélin *et al.*, 2001], ML is used as a post-processor of ST. In [Xiao *et al.*, 2005], ML directly provides a model of the black box program at hand; the test is done by manually inspecting this model.

3 Prior knowledge and Representation

After [Denise *et al.*, 2004], the program being tested is represented from its control flow graph (Fig. 1).

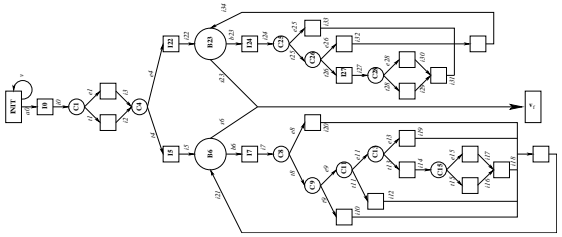

Figure 1: Program FCT4 includes 36 nodes and 46 edges.

Formally, the control flow graph is a Finite State Automaton (FSA) based on some finite alphabet Σ, where Σ includes the program nodes (conditions, blocks of instructions), and the FSA specifies the transitions between the nodes. A program path is represented as a finite length string on Σ, obtained by iteratively choosing a node among the successors of the current node until the final node noted v_f is found.

While the length of program paths is not upper bounded in the general case, for practical reasons coverage-based approaches to software testing consider program paths with bounded length T. Classical results from labelled combinatorial structures [Flajolet *et al.*, 1994] can thus be used to uniformly sample the set of program paths with length in $[1, T]$.

Each path sample is provided to a constraint solver and labelled as feasible or infeasible (see [Denise *et al.*, 2004] and references therein). The infeasibility of a given path arises if it violates some subtle dependencies between different parts of the program. Some general patterns of infeasibility can be identified; due to space limitations, only two most general such patterns will be considered in the rest of the paper.

XOR pattern When two nodes are correlated (for instance two `if` nodes based on an unchanged expression), then their successors are correlated in every feasible path (if the program path includes the `then` successor of the first `if` node, it must also include the `then` successor of the second `if` node). This pattern is referred to as *XOR* pattern, expressing the (possibly long-range) relations between the fragments of the program paths.

Loop(n) pattern The number of times a loop is executed happens to be restricted by the semantics of the problem; e.g. when the problem involves 18 or 19 uranium beams to be controlled, the control procedure will be executed exactly 18 or 19 times [Gouraud, 2004]. This pattern is referred to as *Loop(n)* pattern.

Let us assume that some initial set $\mathcal{E}$ of labelled paths is available, and consider the supervised learning problem defined from $\mathcal{E} = \{(s_i, y_i), s_i \in \Sigma^T, y_i \in \{-1, +1\}, i = 1 \ldots, n\}$, where s_i is a path with length at most T and y_i is 1 iff s_i is feasible.

This learning problem presents some specificities. Firstly, it does not involve noise, i.e. the oracle (constraint solver) does not make errors[2]. Secondly, the complexity of the example space is huge with respect to the number of available examples. In most real-world problems, Σ includes a few dozen symbols; a few hundred or thousand paths are available, each a few hundred symbols long. The number of available paths is limited by the labelling cost, i.e. the runtime of the constraint solver (on average a few seconds per program path). Thirdly, the data distribution is severely imbalanced (infeasible paths outnumber the feasible ones by many orders of magnitude); our attempts for increasing the number of feasible paths through active learning were unsuccessful, as could have been expected from [Dasgupta, 2005]. Lastly, the label of a path depends on its global structure; a Markovian representation, e.g. [Begleiter *et al.*, 2004], would require many more examples to identify the desired long-range dependencies between the transitions.

For these reasons, a frugal propositional representation of strings inspired by Parikh maps [Hopcroft and Ullman, 1979] was considered. For $t = 1 \ldots T$, let $s[t]$ denote the t-th symbol in s, set to value v_f if the length of s is less than t.

- To each symbol v, is associated an integer attribute a_v; $a_v(s)$ is the number of occurrences of symbol v in path s.
- To the i-th occurrence of a symbol v, is associated a categorical attribute $a_{v,i}$. Attribute $a_{v,i}(s)$ gives the next informative[3] symbol following the i-th occurrence of symbol v in s (or v_f if s contains less than i occurrences of v).

4 Overview of *EXIST*

This section presents a new learning framework devised for SST, called *EXIST* for *Exploration vs eXploitation Inference for Software Testing*.

[2] In all generality, three classes should be considered (feasible, infeasible and undecidable) as the underlying constraint satisfaction problem is undecidable. However the undecidable class depends on the constraint solver and its support is negligible in practice.

[3] Formally, $a_{v,i}(s)$ is set to $s[t(i) + k]$, where $t(i)$ is the index of the i-th occurrence of symbol v in s; k is initially set to 1; in case $a_{v,i}$ takes on a constant value over all examples, k is incremented. See [Baskiotis *et al.*, 2006] for more details.

4.1 Position of the problem

While [Denise *et al.*, 2004] proceeds by uniformly sampling the bounded length paths, the efficiency of the approach is limited by the (high) ratio of infeasible paths. Our goal in the present paper will thus be to provide a sampling mechanism, *EXIST*, able to retrieve distinct feasible paths with high probability based on a set $\mathcal{E}$ of feasible/infeasible paths. $\mathcal{E}$ is initially set to a small set of labelled paths, and it is gradually enriched with the paths generated by *EXIST* and labelled by the constraint solver.

This goal differs from that of active learning, aimed at retrieving the most informative examples [Roy and McCallum, 2001]. Active learning is primarily interested in sampling the frontier of the classes to be discriminated, whereas *EXIST* is interested in sampling the feasible class only. Further, as already mentioned active learning is hindered by the imbalanced class distribution [Dasgupta, 2005].

Our goal can also be viewed in the perspective of importance sampling and Estimation of Distribution Algorithms (EDAs; see e.g. [Baluja and Davies, 1998]), aimed at identifying the optima of some fitness function. EDAs iteratively proceed by i) sampling the search space after the current distribution; ii) computing the fitness of the samples; iii) updating and biasing the distribution towards the samples with maximal fitness. The difference relates to the exploration *vs* exploitation trade-off; while EDAs are supposed to gradually switch to exploitation at some point (the distribution converges towards one or a few optima of the fitness function), the *EXIST* goal is to gather examples which are all different. In particular, after a feasible path has been generated, the algorithm must be prevented from generating it again.

The approach can also be compared to on-line learning and the multi-armed bandit problem, where the gambler must choose the next machine to play based on the past selections and rewards [Cesa-Bianchi and Lugosi, 2006; Kocsis and Szepesvári, 2006]. The difference similarly relates to the exploration/exploitation trade-off; while [Cesa-Bianchi and Lugosi, 2006; Kocsis and Szepesvári, 2006] are interested in identifying and running the best policy, we must explicitly avoid repeating our past moves.

4.2 The search space

Borrowing from EDAs the principle of incrementally exploiting and updating a distribution, *EXIST* builds a probabilistic model on top of the extended Parikh map representation (section 3). Formally, the probability of generating a feasible path conditionally to events such as "w is the successor of the i-th occurrence of v and w occurs at least j times", denoted $E_{v,i;w,j}$, is estimated from the set $\mathcal{E}$ of feasible/infeasible paths currently available.

These estimate probabilities are used to gradually construct the current path s, iteratively selecting the successor w of the current symbol v, conditionally to the current number of occurrences of v and w in s.

This formalism is meant to avoid the limitations of probabilistic FSAs and Variable Order Markov Models [Begleiter *et al.*, 2004]. Actually, probabilistic FSAs (and likewise simple Markov models) cannot model the long range dependencies between transitions involved in the prior knowledge (section 3). Variable Order Markov Models can accommodate such dependencies; however they are ill-suited to the sparsity of the initial data available. Rather, the probabilistic Parikh map can be viewed as a most simple case of probabilistic attribute-grammars [Abney, 1997], where the attributes only take into account the number of occurrences of the symbols.

Finally, the *EXIST* system involves two modules: i) the *Init* module determines how the conditional probabilities are estimated; ii) the *Decision* module uses these probabilities to iteratively generate the candidate path. Let us first describe the *Decision* module.

4.3 *Decision* module

Let s denote the path under construction, initialised to the start symbol. Let v denote the last node in s and let i be the total number of occurrences of v in s.

The *Decision* module aims at selecting the successor w of the last node in s. In order to do so, some additional information can be exploited (look-ahead): if the w symbol is selected, then the total number of w symbols in the final path will be at least the current number of occurrences of w in s, plus one; let j_w denote this number.

Formally, the probability $p(s, w)$ of generating a feasible path conditionally to event $E_{v,i;w,j_w}$ (the number of occurrences of v is i and the number of occurrences of w is at least j_w) is provided by the *Init* module; $p(s, w)$ is conventionally set to 1 if there is no path satisfying $E_{v,i;w,j_w}$.

The selection of the next node w aims at maximising the probability of ultimately finding a new feasible path. Three options have been considered:

- The simplest option is the *Greedy* one, that selects the successor node w maximising $p(s, w)$.
- The *RouletteWheel* option stochastically selects node w with probability proportional to $p(s, w)$.
- The *BandiST* option follows the UCB1 algorithm [Auer *et al.*, 2002]; in the standard multi-armed bandit problem this algorithm is shown to achieve logarithmic regret (reward loss compared to the optimal unknown policy) with respect to the number of trials. *BandiST* deterministically selects the node w maximising

$$p(s, w) + \sqrt{\frac{2\,log(e(s, *))}{e(s, w)}}$$

  where $e(s, w)$ is the total number of paths satisfying E_{v,i,w,j_w} and $e(s, *) = \sum_{w\ successor\ of\ v} e(s, w)$.

4.4 The *Init* module

The *Init* module determines how the conditional probabilities used by the *Decision* module are estimated.

The baseline *Init* option computes $p(s, w)$ as the fraction of paths in $\mathcal{E}$ satisfying E_{v,i,w,j_w} that are feasible. However, this option fails to guide *EXIST* efficiently due to the disjunctive nature of the target concept (section 3), as shown on the following toy problem.

Let us assume that a path is feasible iff the first and the third occurrences of symbol v are followed by the same symbol (s feasible iff $a_{v,1}(s) = a_{v,3}(s)$). Let us further assume that $\mathcal{E}$

includes $s_1 = vwvxvw$, $s_2 = vxvwvx$ and $s_3 = vxvwvw$; s_1 and s_2 are feasible while s_3 is infeasible. Consider the current path $s = vwvxv$; the next step is to select the successor of the 3rd occurrence of v. It can be seen that $p(s, w) = .5$ while $p(s, x) = 1.$, as the first event (the 3rd occurrence of v is followed by w and there are at least 2 occurrences of w) is satisfied by s_1 and s_3 while the second event (the 3rd occurrence of v is followed by x and there are at least 2 occurrences of x) only covers s_2.

Therefore, a *Seeded Init* option is devised to remedy the above limitation. The idea is to estimate $p(s, w)$ from a subset of $\mathcal{E}$, called Seed set, including feasible paths belonging to one single conjunctive subconcept.

A necessary condition for a set of positive examples (feasible paths) to represent a conjunctive sub-concept is that its least general generalisation[4] be correct, i.e. it does not cover any negative example. In our toy example problem, the lgg of s_1 and s_2 is not correct as it covers s_3.

***Seeded* Procedure**
$\mathcal{E}^+$: set of feasible paths; $\mathcal{E}^-$: set of infeasible paths
Let $\mathcal{E}^+ = \{s_1, \ldots, s_n\}$ be randomly ordered
Let $E = \{s_1\}$; $h_1 = s_1$
For $t = 2 \ldots n$
  Let $h = lgg(h_{t-1}, s_t)$
  If h is correct wrt $\mathcal{E}^-$
    $h_t = h$; $E = E \cup \{s_t\}$
  Else $h_t = h_{t-1}$
End For
Return E

Figure 2: Construction of a Seed set.

Seed sets are stochastically extracted from $\mathcal{E}$ using the *Seeded* procedure (Fig. 2) inspired by [Torre, 2004]. At the initialisation, the set $\mathcal{E}^+$ of feasible paths is randomly ordered and the seed set E is set to the first path in $\mathcal{E}^+$. Iteratively, one considers the current feasible path s_t and constructs its lgg h with the previously selected paths; if h is correct, i.e does not cover any infeasible paths with respect to the extended Parikh map representation, s_t is added to E. By construction, if the set of infeasible paths is sufficiently representative, E will only include feasible paths belonging to a conjunctive concept (a single branch of the XORs); therefore the probabilities estimated from E will reflect the long range dependencies among the node transitions.

The exploration strength of *EXIST* is enforced by using a restart mechanism to construct another seed set after a while, and by discounting the events related to feasible paths that have been found several times; see [Baskiotis *et al.*, 2006] for more details.

[4]The least general generalisation (lgg) of a set of propositional examples is the conjunction of constraints of the type *[attribute = value]* that are satisfied by all examples. For instance, the lgg of examples s_1 and s_2 in the extended Parikh representation is $[a_v = 3] \wedge [a_{w,1} = v] \wedge [a_{x,1} = v]$.

5 Experimental validation

This section describes the experimental setting used to validate *EXIST*; it reports on the empirical results and discusses the sensitivity of *EXIST* wrt the initial amount of labelled paths.

5.1 Experimental setting and criteria

EXIST is first validated on the real-world Fct4 problem, including 36 nodes and 46 edges (Fig. 1). The ratio of feasible paths is about 10^{-5} for a maximum path length $T = 250$.

A stochastic problem generator has also been designed [Baskiotis *et al.*, 2006] to fully assess the *EXIST* performances. Due to space limitations, three series of results will be presented, related to representative "Easy", "Medium" and "Hard" SST problems, where the ratio of feasible paths respectively ranges in $[5 \times 10^{-3}, 10^{-2}]$ for the Easy problems, in $[10^{-5}, 10^{-3}]$ for the Medium problems, and in $[10^{-15}, 10^{-14}]$ for the Hard problems. The number of nodes varies in $[20, 40]$ and the path length varies in $[120, 250]$. Only 5 *EXIST* variants will be considered, *Greedy* and *SeededGreedy* (*SG*), *SeededRouletteWheel* (*SRW*), and finally *BandiST* (*BST*) and *SeededBandiST* (*SBST*).

For each *EXIST* variant and each problem, the reported result is the number of distinct feasible paths found out of 10,000 generated paths, averaged over 10 independent runs. The baseline (uniform sampling) approach, outperformed by several orders of magnitude, is omitted in the following.

5.2 Results

For a better visualisation, the results obtained by each *EXIST* variant are displayed in parallel coordinates for Easy (Fig. 3), Medium (Fig. 4) and Hard (Fig. 5) problems, starting from a 50 feasible/50 infeasible training set. The average results and the standard deviation of all *Seeded* variants are shown in Table 1.

The computational time ranges from 1 to 10 minutes on PC Pentium 3 GHz depending on the problem and the variant considered (labelling cost non included).

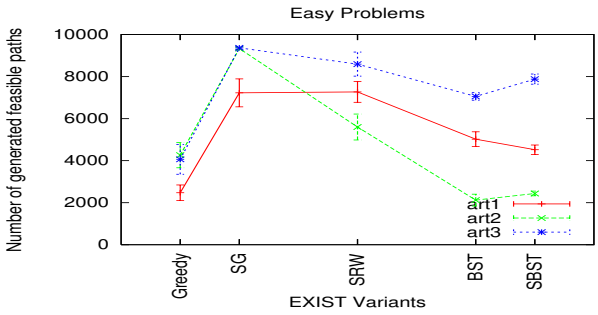

Figure 3: Number of distinct feasible paths generated by *EXIST* out of 10,000 trials on Easy problems, starting from 50 feasible/50 infeasible paths.

In the considered range of problems, the best *EXIST* variant is *SeededGreedy*. On easy problems, *BandiST* and *SeededRouletteWheel* are efficient too (Fig. 3); but their efficiency decreases as the ratio of feasible paths decreases.

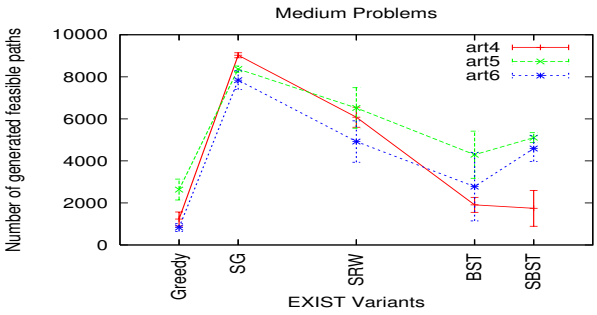

Figure 4: *EXIST* results on Medium problems, with same setting as in Fig. 3.

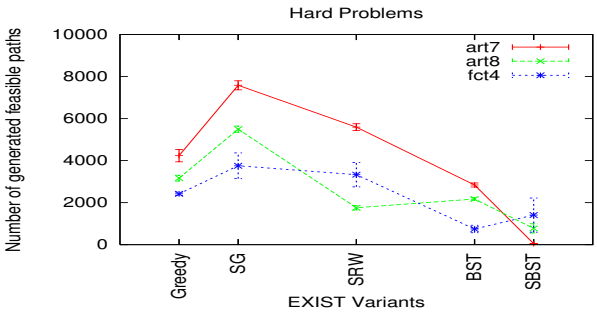

Figure 5: *EXIST* results on Hard problems, with same setting as in Fig. 3.

The *Seeded* option is almost always beneficial, especially so when combined with the *Greedy* and *RouletteWheel* options, and when applied on hard problems. For hard problems, the feasible region is divided among many tiny conjunctive subconcepts. As the probabilities estimated from examples belonging to different subconcepts are misleading, they most often result in generating infeasible examples. Usefully, the *Seeded* option allows *EXIST* to generate infeasible paths which separate the feasible subconcepts. The *Seeded* option is comparatively less beneficial for *BandiST* than for the other options, as it increases the *BandiST* bias toward exploration; unsurprisingly, exploration is poorly rewarded on hard problems.

The comparative performance of *Greedy* and *BandiST* depends on the type of problem. On easy problems, *BandiST* dominates *Greedy*. The reverse is true on hard problems,

Problems		SG	SBST	SRW
Fct4		3754 ± 612	1409 ± 812	3332 ± 580
Easy	art1	7226 ± 665	4520 ± 225	7270 ± 496
	art2	9331 ± 38	2439 ± 110	5600 ± 615
	art3	9365 ± 65	7879 ± 240	8592 ± 573
Medium	art4	9025 ± 118	1744 ± 853	6078 ± 479
	art5	8368 ± 149	5106 ± 236	6519 ± 965
	art6	7840 ± 448	4588 ± 610	4920 ± 984
Hard	art7	7582 ± 217	52 ± 8	5590 ± 163
	art8	5496 ± 149	777 ± 223	1757 ± 110

Table 1: Average number of paths and standard deviation of *Seeded* variants of *EXIST*.

where *BandiST* is dominated due to its exploration bias. Note that both of *BandiST* and *Greedy* will explore a transition which has not yet been explored in the current feasible paths ($p(s, w) = 1$ when there are no paths satisfying the current event, section 4.3). However, if this trial does not lead to a feasible path, *Greedy* will never visit the transition again, while *BandiST* may (and usually will) retry later on.

Note also that *SeededGreedy* displays a significantly lower standard deviation than the other *Seeded* variants (Table 1).

5.3 Sensitivity wrt initial conditions

The usability of *EXIST* would be compromised if a large initial training set were necessary in order to get good results. Therefore experiments with various numbers of initial feasible and infeasible paths (independently selected in $50, 200, 1000$) have been conducted, and the results obtained on a representative medium problem are displayed in Fig. 6.

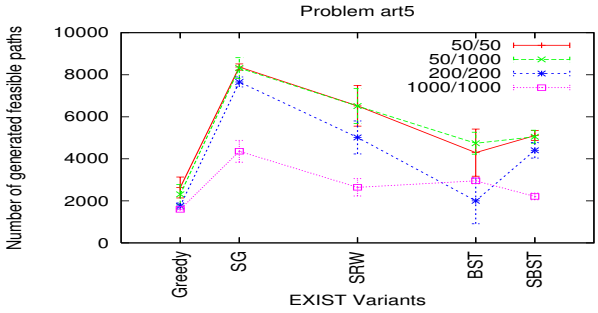

Figure 6: Sensitivity analysis. Average *EXIST* results on some representative Medium problem, depending on the number of initial feasible and infeasible paths.

A first remark is that increasing the number of infeasible paths does not improve the results, everything else being equal. Concretely, it makes almost no difference to provide the system with 50 or 1000 infeasible paths besides 50 feasible paths. Even more surprisingly, increasing the number of feasible paths rather degrades the results (the 1000/1000 curve is usually well below the 50/50 curve in Fig. 6).

Both remarks can be explained by modeling the *Seeded* procedure (Fig. 2) as a 3-state automaton. In each step t, the *Seeded* procedure considers a feasible path s_t, and the resulting lgg is tested for correctness against the infeasible paths. If s_t belongs to the same subconcept as the previous feasible paths (state A), the lgg will be found correct, and the procedure returns to state A. Otherwise (state B), the test against the infeasible paths might reject the lgg (there exists an infeasible path covered by the resulting lgg), s_t is rejected and the procedure returns to state A. Otherwise (state C), there exists no infeasible path enforcing s_t rejection and preventing the overgeneralisation of the seed set. As the seed set will from now on contain examples from different subconcepts (state C is absorbing), the probabilities estimated from the seed set are misleading and will likely lead to generate infeasible paths.

The number and quality of infeasible paths governs the transition from state B to either A (probability q) or C (probability $1 - q$). Although q should exponentially increase wrt

the number of infeasible paths, it turns out that initial infeasible paths are useless to detect incorrect lggs; actually, only infeasible paths sufficiently close (wrt the Parikh representation) to the frontier of the subconcepts are useful. This remark explains why increasing the number of initial infeasible paths does not significantly help the generation process.

On the other hand, the probability of ending up in the absorbing state C (failure) exponentially increases with the number of steps of the *Seeded* procedure, i.e. the number of feasible paths (Fig. 2). Only when sufficiently many and sufficiently relevant infeasible paths are available, is the wealth of feasible paths useful for the generation process.

Complementary experiments ([Baskiotis *et al.*, 2006]) done with 1000 feasible vs 1000 infeasible paths show that i) the limitation related to the number of initial feasible examples can be overcome by limiting the number of feasible paths considered by the *Seeded* procedure (e.g. considering only the first 200 feasible paths); ii) in order to be effective, an adaptive control of the *Seeded* procedure is needed, depending on the disjunctivity of the target concept and the presence of "near-miss" infeasible paths in $\mathcal{E}$.

6 Conclusion and Perspectives

The presented application of Machine Learning to Software Testing relies on an original representation of distributions on strings, coping with long-range dependencies and data sparsity. Further research aims at a formal characterisation of the potentialities and limitations of this extended Parikh representation (see also [Clark *et al.*, 2006]), in software testing and in other structured domains.

The second contribution of the presented work is the *Seeded* heuristics first proposed in [Torre, 2004], used to extract relevant distributions from examples representing a variety of conjunctive subconcepts. This heuristics is combined with Exploration vs Exploitation strategies to construct a flexible sampling mechanism, able to retrieve distinct feasible paths with high probability.

With respect to Statistical Software Testing, the presented approach dramatically increases the ratio of (distinct) feasible paths generated, compared to the former uniform sampling approach [Denise *et al.*, 2004].

Further research aims at estimating the distribution of the feasible paths generated by *EXIST*, and providing PAC estimates of the number of trials needed to reach the feasible paths (hitting time). In the longer run, the extension of this approach to related applications such as equivalence testers or reachability testers for huge automata [Yannakakis, 2004] will be studied.

References

[Abney, 1997] S. P. Abney. Stochastic attribute-value grammars. *Computational Linguistics*, 23(4):597–618, 1997.

[Auer *et al.*, 2002] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[Baluja and Davies, 1998] S. Baluja and S. Davies. Fast probabilistic modeling for combinatorial optimization. In *AAAI/IAAI*, pages 469–476, 1998.

[Baskiotis *et al.*, 2006] N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud. Exploitation / exploration inference for hybrid statistical software testing. Technical report, LRI, Université Paris-Sud, `http://www.lri.fr/~nbaskiot`, 2006.

[Begleiter *et al.*, 2004] R. Begleiter, R. El-Yaniv, and G. Yona. On prediction using variable order markov models. *JAIR*, 22:385–421, 2004.

[Bréhélin *et al.*, 2001] L. Bréhélin, O. Gascuel, and G. Caraux. Hidden markov models with patterns to learn boolean vector sequences and application to the built-in self-test for integrated circuits. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(9):997–1008, 2001.

[Cesa-Bianchi and Lugosi, 2006] N. Cesa-Bianchi and G. Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006.

[Clark *et al.*, 2006] A. Clark, C. C. Florencio, and C. Watkins. Languages as hyperplanes: Grammatical inference with string kernels. In *ECML, to appear*, 2006.

[Dasgupta, 2005] S. Dasgupta. Coarse sample complexity bounds for active learning. In *NIPS*, pages 235–242, 2005.

[Denise *et al.*, 2004] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE*, pages 25–34, 2004.

[Ernst *et al.*, 1999] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999.

[Flajolet *et al.*, 1994] 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.

[Gouraud, 2004] S.-D. Gouraud. *Statistical Software Testing based on Structural Combinatorics.* PhD thesis, LRI, Université Paris-Sud, 2004.

[Hopcroft and Ullman, 1979] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Kocsis and Szepesvári, 2006] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.

[Rish *et al.*, 2006] I. Rish, R. Das, G. Tesauro, and J. Kephart. Ecml-pkdd workshop *automatic computing: a new challenge for machine learning*. 2006.

[Roy and McCallum, 2001] N. Roy and A. McCallum. Toward optimal active learning through sampling estimation of error reduction. In *ICML*, pages 441–448, 2001.

[Torre, 2004] F. Torre. Boosting correct least general generalizations. Technical Report GRAppA Report 0104, Université Lille III, 2004.

[Vardhan *et al.*, 2004] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata. In *FSTTCS*, pages 494–505, 2004.

[Xiao *et al.*, 2005] 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.

[Yannakakis, 2004] M. Yannakakis. Testing, optimization, and games. In *ICALP*, pages 28–45, 2004.

[Zheng *et al.*, 2006] A. X. Zheng, M. I. Jordan, B. Liblit, M. N., and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML*, pages 1105–1112, 2006.