EXIST: Exploitation/Exploration Inference 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

Path-based Statistical Software Testing is interested in sampling the feasible paths in the control flow graph of the program being tested. As the ratio of feasible paths becomes negligible for large programs, an ML approach is presented to iteratively estimate and exploit the distribution of feasible paths.

1 Introduction

This paper is motivated by Statistical Software Testing (SST) and more precisely path-based SST [9]. Path-based SST considers the control flow graph of the program being tested, and samples the feasible paths in this graph (i.e. such that there exists an input case exerting the path); a test set is constructed by gathering all input cases associated to the feasible path samples. However, as the control flow graph is an overly general description of the program, the fraction of feasible paths might be tiny, and it is most often negligible for large program sizes (up to 10^{-5} or 10^{-15}).

This paper presents a generative learning approach aimed at sampling the feasible paths in the control flow graph. This approach, called *EXIST* for *Exploration - eXploitation Inference for Software Testing*, is inspired by both Estimation of Distribution Algorithms (EDAs) [2] and Online Learning [1, 6]. *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 [12] and providing a powerful propositional description of long structured sequences (program paths). Another original contribution, compared to on-line learning [6, 13] or reinforcement learning, is that our goal is to *maximise the number of* distinct *feasible paths* found, as opposed to learning a concept or a fixed policy.

The paper is organised as follows. Section 2 introduces the formal background and prior knowledge related to the SST problem, and describes the extended Parikh representation. Section 3 gives an overview of the *EXIST* system. Section 4 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. Related work are briefly reviewed in Appendix A.

2 Prior knowledge and Representation

The control flow graph of the program being tested 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 (Fig. 1). A program path is 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.

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

Path-based SST relies on classical results from labelled combinatorial structures [11] to uniformly sample the set of program paths with length in $[1, T]$. Each path sample is provided to a constraint solver (oracle) and labelled as feasible or infeasible; see [9] and references therein. The infeasibility of a given path arises if it violates some dependencies between different parts of the program, referred to as *XOR patterns*. For instance if two `if` nodes are 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).

Because of the small number of available labelled paths (due to the labelling cost) compared to the complexity of the "natural" search space, i.e. that of long strings on a large alphabet, a frugal propositional representation inspired by Parikh maps [12] is considered. For $t = 1 \dots T$, let $s[t]$ denote the t-th symbol in string 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[1] symbol following the i-th occurrence of symbol v in s (or v_f if s contains less than i occurrences of v).

Preliminary attempts at discriminant learning have been hindered by the tiny percentage of the feasible paths, as could have been expected from [8]. A generative learning approach was then considered.

3 Overview of *EXIST*

This section describes a sampling algorithm called *EXIST* for *Exploration vs eXploitation Inference for Software Testing*, able to retrieve distinct feasible paths with high probability based on a set $\mathcal{E}$ of feasible/infeasible paths. $\mathcal{E}$, initially set to a small set of labelled paths, is gradually enriched with the paths generated by *EXIST* and labelled by the constraint solver.

EXIST proceeds by iteratively exploiting and updating a probabilistic model $\mathcal{P}$. *EXIST* involves two modules: the *Init* module estimates the probability for a path to be feasible conditionally to its extended Parikh description[2]; the *Decision* module uses the $\mathcal{P}$ model to iteratively construct the current path s.

3.1 *Decision* module

Let s (resp. v) denote the path under construction (resp. the last node symbol in s). Let i be the total number of occurrences of v in s. Let w be one possible successor node of v; if w is selected, 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.

Let us define $p_s(w)$ as the probability for a path S to be feasible conditionally to $E_{s,w}(S) = [a_{v,i}(S) = w] \wedge [a_w(S) \geq j_w]$, estimated by the *Init* module; $p_s(w)$ is conventionally set to 1 if there is no path in $\mathcal{E}$ satisfying $E_{s,w}$.

Probabilities $p_s(w)$ for w ranging over the successors of v are used to select the next node in s. Three options have been considered in order to favor the generation of a new feasible path.
The *Greedy* option 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 considers the multi-armed bandit problem where every bandit arm corresponds to a successor w of the current node v and the associated reward is $p_s(w)$, and uses the UCB1 algorithm [1] for determining the best arm/successor node.

3.2 *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 $E_{s,w}$ that are feasible. However, this option fails to guide *EXIST* efficiently due to the disjunctive nature of the target concept, as shown on the following toy problem.

[1] 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.

[2] This probabilistic model space is meant to avoid the limitations of probabilistic FSAs and Variable Order Markov Models [4]. On one hand, probabilistic FSAs (and likewise simple Markov models) cannot model the long range dependencies of the *XOR patterns*. On the other hand, although Variable Order Markov Models can accommodate such dependencies, they are ill-suited to the sparsity of the initial data available.

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.

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[3] 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.

Seed sets are stochastically constructed as follows. Let $\mathcal{E}^+$ be the randomly ordered set of feasible paths in $\mathcal{E}$. Let the seed set E be initialized to s_1 and let h denote the lgg of elements in E. At step $i \geq 2$, the i-th path s_i in $\mathcal{E}^+$ is considered, the lgg of h and s_i is constructed and its correctness is tested against the infeasible paths in $\mathcal{E}$; if the lgg is correct s_i is added to E. By construction, if the infeasible paths are 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 [3] for more details.

4 Experimental validation

EXIST is empirically validated on a real-world program and on artificial problems. The real-world Fct4 program includes 36 nodes and 46 edges; the ratio of feasible paths is about 10^{-5} for a maximum path length $T = 250$. The artificial problems are derived from a stochastic generator, varying the number of nodes in $[20, 40]$ and the path length in $[120, 250]$ (available on demand from the first author). Three series of results, related to representative “Easy”, “Medium” and “Hard” SST problems are presented in Table 1 and in Fig. 2, 3, 4 (Appendix B). 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. 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 initial $\mathcal{E}$ set includes 50 feasible/50 infeasible paths. 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).

In the considered range of problems, the most robust variant is the *SeededGreedy* one (SG); although *BandiST* and *SeededRouletteWheel* (SRW) are efficient on Easy problems, their efficiency decreases with the ratio of feasible paths. The *Seeded* option is almost always beneficial, especially so when combined with the *Greedy* and *RouletteWheel* options, and when applied on hard problems. 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 sensitivity of the *EXIST* performances wrt the size of the initial training set is studied experimentally, varying the number of initial feasible and infeasible paths in $50, 200, 1000$. The results obtained on a representative medium problem are displayed in Fig. 5 (Appendix B).

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. 5).

Both remarks can be explained by modeling the *Seeded* procedure 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

[3] 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 $[a_v = 3] \wedge [a_{w,1} = v] \wedge [a_{x,1} = v]$.

Problems		Greedy	SG	BandiST	SBST	SRW
Fct4		2419 ± 84	3754 ± 612	745 ± 176	1409 ± 812	3332 ± 580
Easy	art1	2473 ± 372	7226 ± 665	5023 ± 349	4520 ± 225	7270 ± 496
	art2	4261 ± 599	9331 ± 38	2122 ±281	2439 ± 110	5600 ± 615
	art3	4063 ± 711	9365 ± 65	7056 ± 181	7879 ± 240	8592 ± 573
Medium	art4	1235 ± 333	9025 ± 118	1909 ± 358	1744 ± 853	6078 ± 479
	art5	2635 ± 497	8368 ± 149	4294 ± 1121	5106 ± 236	6519 ± 965
	art6	830 ± 187	7840 ± 448	2775 ± 1630	4588 ± 610	4920 ± 984
Hard	art7	4236 ± 292	7582 ± 217	2840 ± 95	52 ± 8	5590 ± 163
	art8	3166 ± 140	5496 ± 149	2174 ± 62	777 ± 223	1757 ± 110

Table 1: *EXIST* variants *Greedy*, *SeededGreedy* (SG), *BandiST*, *SeededBandiST*(SBST) and *SeededRouletteWheel* (SRW): Number of distinct feasible paths out of 10,000 generated paths averaged over 10 independent runs, and standard deviation.

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. Only when sufficiently many and sufficiently relevant infeasible paths are available, is the wealth of feasible paths useful for the generation process.

Complementary experiments 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 $\mathcal{E}$.

5 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 [7]), in software testing and in other structured domains. The second contribution of the presented work is the *Seeded* heuristics inspired by [14], 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 [9].

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 [17] will be studied.

References

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

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

[3] 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.

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

[5] 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.

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

[16] 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.

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

[18] 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.

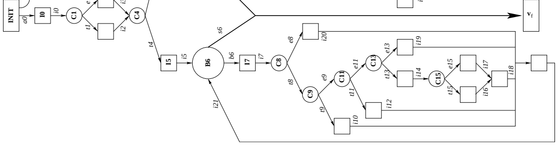

Figure 1: Program FCT4 includes 36 nodes and 46 edges.

A Related Work

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

Ernst et al. [10] 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. [5] 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.

In [15], 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. [16] 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 [18], is actually concerned with software debugging and the identification of trace predicates related to the program misbehaviours.

In [10, 15], ML is used to provide better input to ST approaches; in [5], ML is used as a post-processor of ST. In [16], ML directly provides a model of the black box program at hand; the test is done by manually inspecting this model.

B Empirical validation and sensitivity analysis

Figs. 2, 3 and 4 respectively display the average number of feasible paths out of 10,000 path generations on Easy, Medium and Hard problems.

Fig. 5 displays the average performance of *EXIST* for various numbers of feasible and infeasible paths.

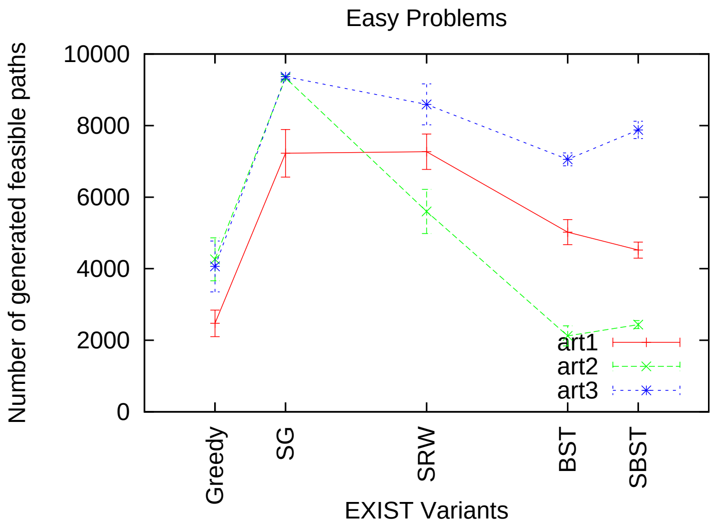

Figure 2: Number of distinct feasible paths generated by *EXIST* out of 10,000 trials on Easy problems, starting from 50 feasible/50 infeasible paths, averaged on 10 independent runs.

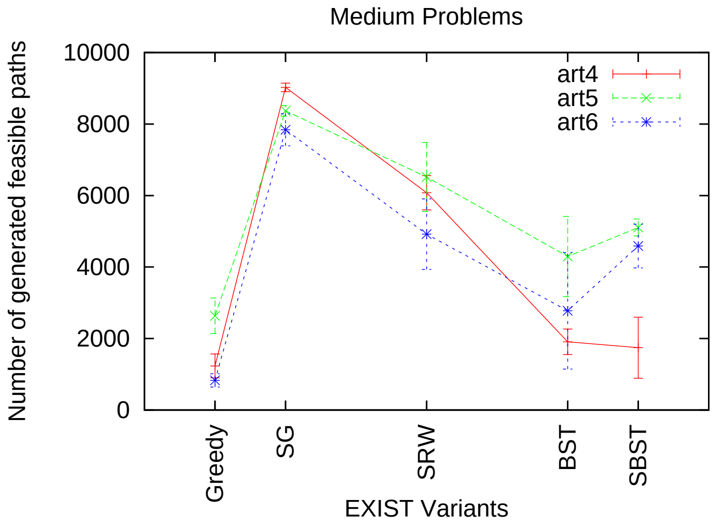

Figure 3: *EXIST* performances on Medium problems, with same setting as in Fig. 2.

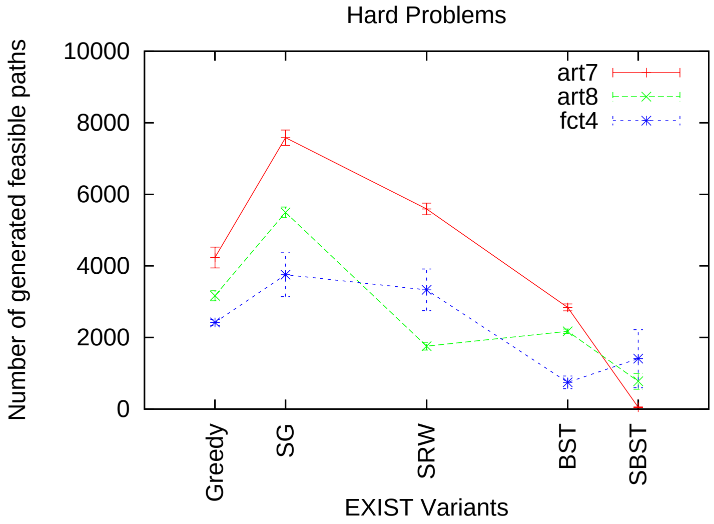

Figure 4: *EXIST* performances on Hard problems, with same setting as in Fig 2.

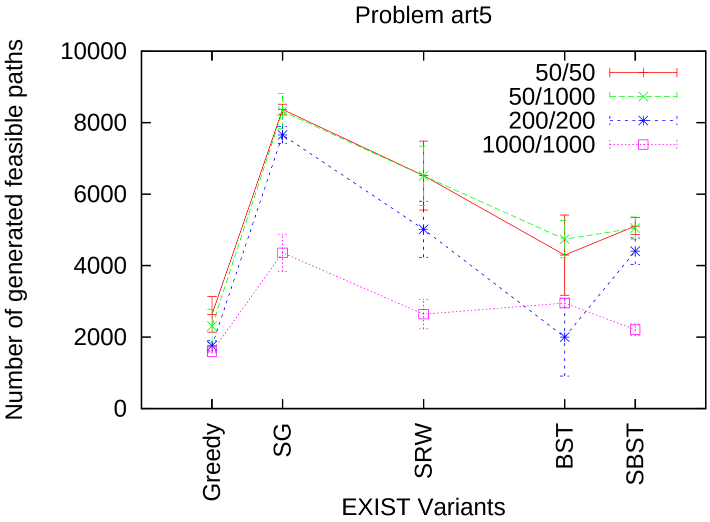

Figure 5: *EXIST* performances on a representative Medium problems, depending on the size and distribution of the initial training set.