Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary

Robert Feldt and Simon Poulding
Dept. of Software Engineering
Belkinge Institute of Technology, Karlskrona, Sweden
Email: robert.feldt@bth.se and simon.poulding@bth.se

***Abstract*—Search-based software testing (SBST) can potentially help software practitioners create better test suites using less time and resources by employing powerful methods for search and optimization. However, research on SBST has typically focused on only a few search approaches and basic techniques. A majority of publications in recent years use some form of evolutionary search, typically a genetic algorithm, or, alternatively, some other optimization algorithm inspired from nature. This paper argues that SBST researchers and practitioners should not restrict themselves to a limited choice of search algorithms or approaches to optimization. To support our argument we empirically investigate three alternatives and compare them to the de facto SBST standards in regards to performance, resource efficiency and robustness on different test data generation problems: classic algorithms from the optimization literature, bayesian optimization with gaussian processes from machine learning, and nested monte carlo search from game playing / reinforcement learning. In all cases we show comparable and sometimes better performance than the current state-of-the-SBST-art. We conclude that SBST researchers should consider a more general set of solution approaches, more consider combinations and hybrid solutions and look to other areas for how to develop the field.**

I. Introduction

The term Search-Based Software Testing (SBST) describes a number of powerful methods that permit practitioners to generate test suites for efficiently, and effectively, testing functional and non-functional aspects of software systems [1], [2], [3]. If the testing goals can be quantified in some manner—or, in some cases, simply ordered—a multitude of different search techniques can, in principle, be applied in these methods. However, in practice, both SBST practitioners and researchers typically use some form of evolutionary search, i.e. search algorithms inspired by evolution and natural selection. We fear that this might limit results and stifle creativity. In this paper we urge SBST researchers to broaden their view of what the search aspect of SBST can mean and we will show that many alternative techniques for search and optimization might have value.

In order to better understand the current application of search algorithms in SBST, we analyzed recent testing papers from three of the field's major conferences: the International Symposium on Search-Based Software Engineering (SSBSE), the search-based software engineering track at the Genetic and Evolutionary Computation Conference (GECCO), and the International Workshop on Search-Based Software Testing (SBST). A total of 39 empirical or tool papers on SBST were published at these venues in 2013 and 2014. Two-thirds of the papers—26 out of 39—applied an evolutionary algorithm, of which 23 applied a Genetic Algorithm (GA): 15 as a standard GA, 2 as a GA-based memetic algorithm, and 7 as a multi-objective GA (the majority using NSGA-II)[1]. The next most-frequently applied algorithms were Genetic Programming (4 papers), (1+1) EA (4 papers), hill-climbing (3 papers), and alternating variable methods (3 papers). Our analysis suggests that evolutionary search, and GAs in particular, are the algorithms of choice for both single- and multi-objective problems in SBST.

We offer a number of explanations for this prevalence of GAs as the search technique. GAs can be applied to a wide range of problem classes and typically find solutions with acceptably good quality. This wide applicability permits us, as researchers, to re-use the knowledge gained in applying GAs to one testing problem when solving subsequent problems. In addition, there is a great deal of active research in GAs that can guide their application to testing problems, and this research is typically disseminated in a form that is readily-accessible to us. In contrast, the research on classic optimization algorithms is often described for fellow mathematicians and may be less accessible.

Nevertheless we argue that there are disadvantages to this 'one-size-fits-all' approach to choosing the search algorithm. While GAs may provide acceptable performance, other algorithms may be better at exploiting the particular structure of a testing problem: the focus on GAs may be limiting our creativity as researchers in finding much better fits between the testing problem and the search algorithm. In particular, GAs can scale poorly to high-dimensional problems and scalability is sensitive to the representation of the problem (see, for example, [4]). Moreover, GAs have many parameters that must be tuned to the specific problem if the best performance is to be realised.

This paper makes the three major contributions:

- An argument that SBST researchers should broaden their horizons and consider search techniques other than evolutionary algorithms.
- An empirical case study that supports this argument by demonstrating that three alternative classes of al-

[1]Some papers applied more than one algorithm and so are counted in more than one category.

gorithm can have advantages over evolutionary search: 'traditional' non-evolutionary algorithms of the type used in Operations Research; a machine learning algorithm: Gaussian Processes; and Nested Monte-Carlo Search, a form of Monte-Carlo Tree Search from the domain of automated game-playing.
- Guidelines for SBST researchers when choosing and investigating search techniques as part of their testing methods.

Section II below describes the method we used in the subsequent sections each focusing on one of the three alternatives to evolutionary search in SBST: traditional optimization algorithms (Section III), machine learning (Section IV), and Nested Monte-Carlo Search (Section V). Then, Section VI discusses the results, and Section VII concludes.

II. Method

The goal of the empirical work is to consider multiple alternatives to an evolutionary search algorithm for one and the same search-based software testing problem. However, to evaluate the advantages and disadvantages of each approach we need to be able to vary the difficulty and size of the problem; this should make it easier to highlight any differences in performance, scalability or robustness.

Thus, we have selected three different test data generation problems of differing size but each using the same SBST tool: the GödelTest automated test data generation system [5], [6]. For the purposes of this paper the details of GödelTest are not critical; we refer the reader to earlier papers for details about the system. However, we give a very brief introduction to the key concepts of the system below so that we can then describe the specific SBST problems we have selected.

GödelTest allows a tester to write data generators in the form of program code and to set specific goals for properties that the data, or its use in actual testing, should have. The generators constrain the ways in which a datum can be generated while allowing a large range of data to be generated depending on random choices—at locations in the program called 'choice points'—during the generation process. By selecting a class of models for how these choices can be made (a choice model) and then searching for a specific model instantiation from which good data are generated, the system can help the tester fulfill specific testing goals. Note the clear benefits of a search-based approach to this problem. The system does not care what the properties are as long as they give a numeric indication of whether a good or bad datum has been generated. The properties can be specific to the datum itself (its length/size or depth for example, or its uniqueness compared to already seen data), to a set of data, or be properties measured during execution of the software-under-test (SUT) while providing the datum as input.

To eliminate variation we use only one fitness criteria for all of the investigated GödelTest test data generators. The goal is to create test data of specific goal sizes, measured individually on each datum. Even though GödelTest could as well use branch or statement coverage or even a mutation testing score, measuring the size of the data is simpler and quicker. Since we will perform experiments with many different optimization algorithms we do not want to need to start and actually run a SUT a large number of times. We thus limit ourselves to a fitness property that can be measured directly on a generated datum and vary difficulty by the size of the generators and choice model classes involved.

From a total of three different generators and two different choice model classes we have created six different optimization problems in the empirical work of this paper. Table I summarizes the six different problems and their main characteristics.

Problem 1 (Expr in the table) uses the default sampler choice model (called simply DefaultCM in the following) in GödelTest for a recursively defined generator for simple arithmetic expressions. DefaultCM maps each choice point in a generator to a specific statistical distribution. Which distribution to use is typically theoretically motivated. For example, the number of repetitions to use for repeat choice points is selected from a Geometric distribution. This distribution has for example been used in the theoretical work on Boltzmann samplers [7] in order to guarantee that all possible recursively defined trees (up to a given size) can be sampled. For rule choices (which occur when there are multiple rules to select from) we naturally use a categorical distribution, and so on for other choice point types. Given that there are 5 choice points in the expression generator used in problem one and some distributions need two instead of one real-valued parameter(s) there is a total of 8 reals that can be selected during the search.

Problem 2 (ExprMix) is the same generator as in problem 1 but the sampler choice model use is the default one but with the repetitions choice points mapped to a mixture sampler that, with a certain probability, either selects the default geometric distribution, or, alternatively, selects a gaussian distribution with a specific mean and standard deviation (sigma) value. Thus the number of decision variables for problem 2 is larger, and requires the optimization algorithms to tune 12 different reals.

Problems 3 (LibXml) and 4 (LibXmlMix) uses the same choice model classes (DefaultCM and MixtureCM, respectively) but combines them with a generator for XML documents of a small library, i.e. to model the meta-data of books and loans. They have 81 and 109 decision variables, respectively. Finally, problems 5 (MathML2) and 6 (MathML2Mix) similarly combine the two mentioned choice model classes with a larger generator of XML documents, namely for the MathML2 standard. They have 360 and 492 decision variables, respectively. The generators used in problems 3&4 and 5&6 have both been automatically generated from XSD specifications as described in our earlier paper [8].

A. Experiment settings and fitness function

For all problems we have executed 10 repeated runs per optimization algorithm per goal. Repeated runs are needed since all the algorithms are stochastic and their results will depend on non-deterministic choices during their execution.

TABLE I
THE SIX SBST OPTIMIZATION PROBLEMS USED IN THE EMPIRICAL WORK AND THEIR MAIN CHARACTERISTICS

Id	Name	Generator	Choice model	Decision variables	Goals (Len. of gen. test data)
1	Expr	Recursively defined expressions	Default	8	50, 200
2	ExprMix	Recursively defined expressions	Default w. mixture sampler	12	50, 200
3	LibXml	Library XML	Default	81	100, 500
4	LibXmlMix	Library XML	Default w. mixture sampler	109	100, 500
5	MathML2	MathML2 XML	Default	360	100, 1000
6	MathML2Mix	MathML2 XML	Default w. mixture sampler	492	100, 1000

With 10 repeated runs we can get an indication of average results without an excessive number of repetitions. More repetitions would be needed to compare algorithms on individual problems though. All optimization algorithms were used with their default settings. Better results can often be had by tuning parameters for individual algorithms; however this is out of the scope of this paper.

While running the optimizations we save each fitness improvement to disk. This way we can later graph and compare algorithm performance per the same number of fitness evaluations. The fitness function for is often noisy since the stochastic GödelTest generators emit data with a wide range of lengths, so we sample a set of 200 (N) data from a generator for each fitness calculation. The specific fitness function used is a weighted sum of the mean absolute percentage error (MAPE) and the percentage of invalid data generated:

$$f(m_i) = \sum_{j=1}^{N} \frac{100 * |Len(d_j) - G|}{N * G} + 10 * \frac{100 * \#\text{ invalid datums}}{N}$$

where $Len(d_j)$ is the length of a specific generated datum when dumped as a string, m_i is the specific generator and model being evaluated, N is the number of sampled data, and G is the current goal value. A datum is counted as invalid if more than 2000 choices need to be made while generating it or if a repetition choice point requests more than 1000 repetitions. These limits are ad hoc, but put in place to limit execution times. They were selected high enough so as not to negatively affect optimization performance, based on initial experiments.

We tried a few different weights but it did not seem to have a big impact on performance and thus settled on a value of 10. We selected MAPE in the fitness function since that makes it easier to compare results for different optimization problems and goals. All experiments were executed on an Apple MacBook Pro Retina (mid 2012) with a 2.7MHz Intel Core i7 (4 CPU cores), and 16GB 1600MHz DDR3 RAM and using Julia v0.3.5 (2015-01-08). Only one core was used per optimization algorithm but up to 4 runs were executed in parallel for maximum CPU utilization. The stopping condition for each run was 2000 fitness evaluations (when using the alternative optimization algorithms). The results in the paper is based on sampling 1000 data from the best model found by an optimization algorithm and calculating a number of summary statistics for this set.

III. ALT 1: ALTERNATIVE OPTIMIZATION ALGORITHMS

Optimization is a widely applied and beneficial tool in the sciences and engineering and has been studied for a long time in a number of different communities, e.g. mathematical programming, operations research and non-linear optimization, to name but a few. The Nelder-Mead simplex local optimization algorithm was proposed in the 1960s [9] and Brent proposed the PRAXIS local optimization algorithm in his book from 1973 [10]. Most interest has been traditionally given to methods that require the fitness function to be differentiable. However, this is rare in SBST. Luckily there are many application areas with this characteristic so there has been steady progress also on so called direct search methods (called derivative-free methods by others). For more detailed introductions with many example algorithms see [11], [12].

We do not have space here for an extensive evaluation of the many possible algorithms but have selected a (convenience sampled) subset. A major criterion was that an open-source and well documented library implementing them was available. Since our SBST tool is implemented in Julia we preferred libraries that can interface directly with this programming language. Since SBST often requires long-running and many fitness evaluations the implementations also needs to be performant.

Fortunately, the NLopt open-source library fulfills all our criteria: it is implemented in C with interfaces to many modern-day programming languages including Julia, includes many classic algorithms with up-to-date implementations based on recent tweaks proposed in published research, and it is relatively well-documented. It includes both local and global optimization algorithms as well as tutorials and concrete advice on how to combine algorithms together. A typical recommendation is to first run a global optimization algorithm and then "polish off" the best solution found with a shorter local optimization run. We followed this recommendation for the 3 algorithm combinations from NLopt we used in the empirical work: CRS+BOB, ESCH+COB, and SRES+BOB. Additionally, we included the traditional differential evolution (DE) algorithm used in our previous papers on GodelTest [5] as implemented in the BlackBoxOptim Julia package developed by one of the authors. We also included a random search algorithm (RandSrch) and a classic, direct search algorithm (Compass) that were also available in BlackBoxOptim. Some further details about the evaluated algorithms can be found in

TABLE II
OPTIMIZATION ALGORITHMS USED AND THEIR MAIN CHARACTERISTICS

Name	Description	Refs	Type	Library
DE	Differential evolution (rand/1/bin)	[13]	Evo	BlackBoxOptim
SRES+BOB	Stochastic Ranking Evo Strategy, then BOBYQA	[14], [15]	Evo. Combination	NLopt
ESCH+COB	Evo Strategy, then COBYLA	[16], [17]	Evo. Combination	NLopt
CRS+BOB	Controlled Random Search (CRS), then BOBYQA	[18], [19], [15]	Combination	NLopt
Compass	Direct (compass) search with adaptive step size	[11]	Non-Evo	BlackBoxOptim
RandSrch	Random Search (baseline)	N/A	Non-Evo	BlackBoxOptim

TABLE III
SUMMARY RESULTS FOR INVESTIGATED OPTIMIZATION ALGORITHMS ON EACH OF TWO GOALS FOR THE SIX INVESTIGATED SBST PROBLEMS. VALUES LISTED ARE THE AVERAGES FOR 120 RUNS OF EACH OPTIMIZATION ALGORITHM. RANK WAS CALCULATED BASED ON LOWEST ACHIEVED MAPE VALUE AFTER ONE RUN OF EACH OF THE 7 INCLUDED ALGORITHMS (THEN AVERAGED), THE OTHER COLUMNS SHOWS THE AVERAGE FOR ALL RUNS PER ALGORITHM, EXCEPT WINS THAT SHOW HOW MANY TIMES (OF 12) THE ALGORITHM HAD THE LOWEST MAPE VALUE.

Algorithm	Rank	Wins	MAPE	HitRate	Time
ESCH+COB	**2.2**	**6**	**29.7**	5.5	**705.3**
CRS+BOB	3.21	4	44.5	**8.5**	1203.7
DE	3.24	2	43.6	4.0	985.9
RandSrch	3.8	0	46.9	1.8	1145.3
Compass	3.9	0	49.3	2.1	1168.6
SRES+BOB	4.9	0	64.3	3.5	1583.1
Unoptimized	6.7	0	164.79	0.2	0.0

table II and the table also refers to papers with more details for each algorithm.

Table III summarizes all of the results from 120 executions of each of the evaluated optimization algorithms (720 runs in total with a total of ∼121 CPU hours on a single-core processor). It also includes the same results from a 1000 data sampled from the unoptimized generator for each problem. This shows a baseline for the performance one can get on these problems by using GodelTest right out of the box. The RandSrch algorithm shows another type of baseline: what a random search using the same (2000) number of fitness evaluations would give. For each algorithm the table lists the average rank, the MAPE, the hit rate (percentage of generated data that are exactly on the goal value), and the time needed for one optimization run. For example we can see that the ESCH+COB combination from the NLopt library is on top based on an average rank of 2.2, with a MAPE of 29.7% from the goal value, returns a datum with exactly the goal characteristic about 5.5% of the time after an average optimization time of 705.3 seconds.

Overall we see that some optimization is needed since the MAPE for the unoptimized generators is on average ∼165% off target. By spending a couple of minutes of optimization time the best performers can take this down to ∼20-40% (depending on problem) while increasing the hit rate from 0.2% up to ∼5-8%. Note that the average time presented here is negatively affected by problems 3 and 4 which took a very long time for all the algorithms; for the other problems the optimization time was just a few minutes. There is no clear pattern that any specific type of optimization algorithm have an edge; even though the ESCH+COB hybrid algorithm are in top both on rank, MAPE and time needed performance varied over the problems. In fact the CRS+BOB non-evolutionary hybrid had the largest number of top ranked positions but was negatively affected on average because of bad performance on problems 5 and 6; we can only speculate that the random search component of CRS does not scale as well when the number of dimensions increase. Also, CRS+BOB had consistently higher hit rates than the others except for problems 5 and 6. Random search also generally performs relatively good but has a lower hit rate. Possibly, this can be explained by the fact that a random search might stumble upon solutions that are in the vicinity of the goal but then cannot refine the solutions to approach the goal further. This analysis is supported by the fact that CRS+BOB is essentially a smarter way to randomly search throughout the search space and then uses the BOBYQA local optimization algorithm to fine-tune the best solution found.

We also looked at robustness of the optimization algorithms. CRS+BOB has a somewhat larger variation in MAPE than the other algorithms. Overall, we also note that there is less difference for the problems using the default sampler. Probably this model (class) is too restrictive while the use of a mixture sampler allows more fine-tuned adaptation. Future work should investigate if a better default model should be included with the system.

When it comes to time performance the evolutionary algorithms seem to have a slight edge with ESCH+BOB the fastest and DE second (but 40% slower on average). Here CRS+BOB seem to take 30-50% more time on average, depending on problem. Our analysis is that since it uses low-discrepancy Sobol sequences to ensure a better coverage of the whole search space it will also sample more points that lead to many choices during generation and thus long fitness evaluation times. The evolutionary algorithms will rather focus performance on more promising areas of the search space where good performance can be found which also generates fewer invalid data. This is not always the right trade-off; depending on problem characteristic the CRS approach may sometimes be preferable.

The top performer both in sub-classes based on different choice model classes as well as overall, i.e. ESCH+COB, uses a combination with 90% of evaluations given to an evolutionary algorithm and then fine-tuning the best solution found via a short run of the COBYLA local optimization algorithm. Maybe such hybrid solutions are worth investigating further in SBST more generally? It seems to be both robust, time efficient and gives top performance on all problems we investigated.

IV. Alt 2: Optimization via Machine Learning

Machine Learning is a very active area of research and has seen a plethora of results in the last 15-20 years. The area is broad and concerned with many different aspects of how computers can better learn from data and create predictive models. One of the important contributions in recent years are Gaussian Processes (GPs), a way to put a prior probability on a set of functions and then update the probabilities in a bayesian framework to adapt the functions to collected data [20]. GPs are powerful kernel-based methods and can model complex, non-linear functions in a non-parametric fashion. Also since they model the uncertainty of their estimates they provide additional information which is commonly utilized to improve optimization of simulation models and algorithm hyper-parameters [21]. A downside is that they typically do not scale well to problems with very many decision variables.

We applied GPs to our problem by using it as a surrogate function for modeling the fitness landscape. Starting from an initial 25 points sampled via latin hypercube sampling we evaluated their fitness and used the GP for regressing the fitness value from the decision variables. We then sampled a large number of random points as well as random points at different step sizes away from the current minima. The GP then predicted the fitness of all of these candidates and the top 5 were then actually evaluated on the fitness function. Based on the new points we then updated the GP surrogate function and repeated.

Space does not permit a detailed analysis, but after only 2000 sampled datums this approach could reach MAPE levels of $\sim$80% on problem 5 and after 20000 sampled datums it frequently reach MAPE levels of $\sim$50%. This does not rival the top performer for this problem (ESCH+BOB) that reached on average a MAPE of $\sim$39% after 400,000 sampled datums (2000 evaluations with 200 datums per evaluation) but the key point is to quickly get a rough sense of the shape of the fitness landscape. Even on this high-dimensional problem a GP approach can provide a quick focusing of interesting areas of the search space. However, when running for longer time the GP is not a practical solution since a vanilla implementation of it, such as ours, scales like $O(N^3)$ in the number of regressed points the modeling time quickly becomes excessive. However, it is interesting to consider them as being at one extreme of a spectrum of optimization approaches; they can quickly sketch out the overall shape of a fitness landscape. For example, we envisage them being useful for identifying areas of a landscape that lead to long fitness evaluation times, which can then be avoided.

V. Alt 3: Nested Monte-Carlo Search

Monte-Carlo Tree Search (MCTS) methods use stochastic simulation (the 'Monte-Carlo' aspect) to efficiently solve problems that can be represented as finding the best path of choices through a tree of decisions [22]. For each possible choice at a decision point, one or more random simulations are performed in order to assess the impact of taking that choice, and this information is used by the search to select one of the choices. Although the assessment through random simulation will necessarily be noisy, there is often sufficient signal to guide the search along a good path. MCTS methods have demonstrated particular success in the domain of automated game play, the tree of decisions in this domain being the possible moves made at each turn in the game by the automated player and one or more human opponents.

In our previous work [6], we demonstrated that Nested Monte-Carlo Search (NMCS), a variant of MCTS applied to single-player games [23], could be applied to the generation of test data by GödelTest. The tree of decisions in this context are the choice points encountered when executing the generator program. The algorithm is described in detail in [6]; we summarize it here in conjunction with Figure 1. The current choice point has three choices. For each of these choices, a single random simulation is performed that first takes that choice and then subsequently makes random choices—sampled from a choice model such as the default sampler choice model—until the execution of the generator is complete (indicated by the filled node). The properties of the datum generated by each simulation are evaluated, and the choice that led to the best datum is taken. The process is then repeated at the next choice point encountered by the generator.

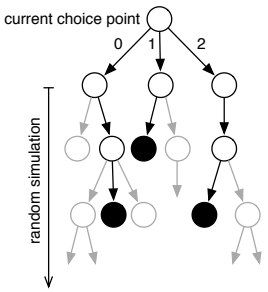

Fig. 1. Nested Monte-Carlo Search applied to GödelTest.

Our motivation for applying NMCS to GödelTest was that—in contrast to optimizations applied to the choice model—NMCS exerts control over choices made *during* the generation process itself. The random nature of the simulations made by NMCS ensures that the data generated across multiple runs of the generator remains diverse, but we speculated that NMCS could dynamically adjust the path taken in the generator to ensure that the each generated datum had desired properties. The empirical results of our previous work [6] support this hypothesis.

However the effectiveness of NMCS was found to be sensitive to the choice model used for the random simulation. For that reason we demonstrate here a combination of NMCS and a search algorithm: the choice model is first optimized by a search algorithm, and then NMCS is used to sample test data using the optimized choice model with the objective of generating test data very close to the desired properties.

In Table IV we show the results of combining NMCS with choice models that have been pre-optimized with one of the top 3 optimization algorithms from Section III above. We also include the random search as well as 100 samples from the (unoptimized) sampler choice model as baselines. The goal here is to create 100 MathML2 XML documents each having a length of 1000 characters. The results are based on a very brief pre-optimization step with only 400 fitness evaluations (each using a sample of 50 data) since only very little time would typically be available for pre-optimization for a specific testing target.

The result that stands out is the effectiveness of NMCS itself. There is some advantage in combining NMCS with pre-optimized choice models, but since NMCS gets very low MAPE values even when combined with the unoptimized model, the benefit is limited. On average there is a speedup but the average generation time when NMCS is combined with one of the pre-optimized models is larger than when using the default model. And since the speed-ups (here measured in relation to Unoptimized+NMCS) are still modest compared to the time it takes to run a pre-optimization we doubt such combinations have much value in practice. The main benefit from NMCS comes from the algorithm itself but since it can help to pre-optimize, future work should investigate doing this for stretch goals that are very hard to reach.

VI. Discussion

Our results confirm that evolutionary search is not always the best way to perform the search part of search-based software testing. There are alternative algorithms from the optimization literature that can give similar or sometimes better results. By reconsidering what the search is used for we can find alternative approaches that are not purely focused on optimization. Gaussian processes (GPs) from machine learning can quickly find roughly the right areas of the search space with very few fitness evaluations and nested monte-carlo search (NMCS) from reinforcement learning can hone in more precisely on the testing target one has set. And throughout our investigations we have repeatedly seen that random search often can perform on par with more advanced approaches. Our empirical results are only indicative; more experiments are needed before we can conclude that any one approach has a particular advantage. It is also not clear if the selected test generation problems are representative. We also note that future research should investigate also alternative approaches in an even wider sense of the word; most of the algorithms investigated here can still be considered 'soft computing'. Below we note some implications of this work and guidelines that we set for ourselves in future SBST work. We urge other SBST researchers to consider if they should adopt something similar for their work.

Evolutionary algorithms are very general search algorithms and rarely fail outright when applied on difficult search problems. It is understandable that they have gained a strong following and are the 'workhorse' of SBST. However, one downside is that they often have a plethora of tuning parameters. In contrast, many of the traditional optimization algorithms included in for example the NLopt library have no or only a few tunable parameters. They have been around for long and often performs well without tuning. Similar arguments have been made in SBST and, for example, Arcuri and Fraser, after an extensive empirical investigation, concluded that default settings for parameters are often a sensible choice since tuning might not pay off [24]. However, this depends on how well the developer of the search algorithm have tuned parameters and your context might differ from theirs. For ultimate control we often implement our own search algorithms and thus must tune their parameters; algorithms that require less or no tuning would thus be preferable.

A subtle example of these problems have cropped up during the empirical work for this paper. We had tuned the general DE algorithm used by default in GödelTest for long optimization runs which thus can start from a large population to better retain diversity. However, for the practical testing work for which we develop GödelTest, what a tester needs is to more quickly find a spread of testing data with differing characteristics. Thus, we might be better off with multiple short runs starting from very small populations. Or we can consider a simple alternative such as random search or machine learning approaches that can extract more information per evaluation, and thus more quickly can find roughly the right place to be. Or start from one such 'rough' approach and then refine it with a local search as implemented in libraries like NLopt.

When we applied Nested Monte-Carlo Search to our problem we had to rethink what was actually our goal with the search and we had to view our problem in a new light. The approach has similarities with Estimation of Distribution Algorithms (see for example, [25]), the evolutionary algorithms that build stochastic models of which parts of and values in the genotype give good performance for the phenotype. But NMCS is more direct and thus simpler to implement. It does not need to build up the model, or implement specific data structures to house such a model, by simply randomly sampling new paths, or falling back to a previous good choice, it can find a good way forward. But NMCS has also made us rethink our problem because of its very different characteristics; rather than being 'generally' good but not ever exactly right (like DE) it takes a much longer time to find something very close to target. This is not a first choice of search algorithm in a testing tool; for complex data generators the testers might have to wait a long time to even get one test datum. But by combining NMCS with an initial random or maybe GP search over the search space a good complement might be NMCS to seek out specific target areas where failures might be lurking. And against such a combination

TABLE IV
RESULTS WHEN PRE-OPTIMIZING A CHOICE MODEL AND THEN USING IT WITHIN THE NESTED MONTE CARLO SEARCH CHOICE MODEL THAT ADAPTIVELY SELECTS PATHS THAT LEAD CLOSER TO A TARGET MATHML2 XML OUTPUT LENGTH OF 1000 CHARACTERS.

Algorithm combination	Mean length	MAPE	HitRate	Avg. Time	Avg. Speedup
ESCH+COB+NMCS	**997.2**	**0.3**	97.3	7.4	1.4
RandSrch+NMCS	996.5	0.4	97.6	3.5	2.2
DE+NMCS	996.4	0.4	98.5	2.9	3.2
CRS+BOB+NMCS	996.4	0.6	**98.8**	1.7	1.8
Unopt+NMCS	983.6	1.6	96.9	2.2	1.0
Unoptimized	257.0	73.3	0.0	**0.1**	**35.8**

it is less clear what are the key advantages of an evolutionary search. During this work we have come to view our DE as a generalist without much teeth. We end this discussion by listing reminders to ourselves in future SBST work:

- **Investigate alternatives** - by adopting your problem to an alternative technique you change your view on it and avoid limiting your creativity, and
- **Combine approaches** - hybrids can combine strengths of different members into a more adapted whole, and
- **Always include random search** - since you cannot know in advance if your problem really benefits from a more refined algorithm, and you will learn from the sanity check, and
- **Consider existing libraries** - it is not always best to implement yourself, lots of work has gone into debugging and tuning parameters in existing optimization libraries.

VII. CONCLUSION

The empirical work in this paper illustrates an SBST method for which search techniques other than an evolutionary algorithm can achieve better results. This supports our more general position: that researchers in SBST should avoid the 'one-size-fits-all' approach to selecting the search technique to use by considering a wider class of optimisation algorithms, such as traditional Operations Research algorithms, approaches from machine learning, and hybrid approaches.

ACKNOWLEDGMENT

This work was funded by The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC).

REFERENCES

[1] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[2] ——, "Search-based software testing: Past, present and future," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 153–163.

[3] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[4] D. Thierens, "Scalability problems of simple genetic algorithms," *Evol. Comput.*, vol. 7, no. 4, pp. 331–352, Dec. 1999.

[5] R. Feldt and S. Poulding, "Finding test data with specific properties via metaheuristic search," in *Proc. of 24th IEEE International Symposium on Software Reliability Engineering (ISSRE 2013)*. IEEE, 2013, pp. 350–359.

[6] S. Poulding and R. Feldt, "Generating structured test data with specific properties using Nested Monte-Carlo Search," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014)*, 2014, (to appear).

[7] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer, "Boltzmann samplers for the random generation of combinatorial structures," *Combinatorics, Probability and Computing*, vol. 13, no. 4-5, pp. 577–625, 2004.

[8] S. Poulding and R. Feldt, "The automated generation of human-comprehensible XML test sets," in *Proc. 1st North American Search Based Software Engineering Symposium (NasBASE)*, 2015, to appear.

[9] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[10] R. P. Brent, *Algorithms for minimization without derivatives*. Courier Dover Publications, 1973.

[11] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM review*, vol. 45, no. 3, pp. 385–482, 2003.

[12] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to derivative-free optimization*. Siam, 2009, vol. 8.

[13] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[14] T. P. Runarsson and X. Yao, "Search biases in constrained evolutionary optimization," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 35, no. 2, pp. 233–243, 2005.

[15] M. J. Powell, "The bobyqa algorithm for bound constrained optimization without derivatives," *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge*, 2009.

[16] C. H. S. Santos, M. S. Goncalves, and H. E. Hernandez-Figueroa, "Designing novel photonic devices by bio-inspired computing," *Photonics Technology Letters, IEEE*, vol. 22, no. 15, pp. 1177–1179, 2010.

[17] M. J. Powell, "A direct search optimization method that models the objective and constraint functions by linear interpolation," in *Advances in optimization and numerical analysis*. Springer, 1994, pp. 51–67.

[18] W. Price, "Global optimization by controlled random search," *Journal of Optimization Theory and Applications*, vol. 40, no. 3, pp. 333–348, 1983.

[19] P. Kaelo and M. Ali, "Some variants of the controlled random search algorithm for global optimization," *Journal of optimization theory and applications*, vol. 130, no. 2, pp. 253–264, 2006.

[20] C. Rasmussen and C. Williams, "Gaussian processes for machine learning," *Adaptive computation and machine learning*, 2006.

[21] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," *arXiv preprint arXiv:0912.3995*, 2009.

[22] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[23] T. Cazenave, "Nested Monte-Carlo search." in *Proc. 21st Int'l Joint Conf. Artificial Intelligence (IJCAI)*, 2009, pp. 456–461.

[24] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[25] P. Larrañaga and J. A. Lozano, *Estimation of distribution algorithms: A new tool for evolutionary computation*. Springer Science & Business Media, 2002, vol. 2.