Generating Structured Test Data with Specific Properties using Nested Monte-Carlo Search

Simon Poulding
University of York
York, YO10 5GH, UK
simon.poulding@york.ac.uk

Robert Feldt
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
robert.feldt@bth.se

ABSTRACT

Software acting on complex data structures can be challenging to test: it is difficult to generate diverse test data that satisfies structural constraints while simultaneously exhibiting properties, such as a particular size, that the test engineer believes will be effective in detecting faults. In our previous work we introduced GödelTest, a framework for generating such data structures using non-deterministic programs, and combined it with Differential Evolution to optimize the generation process.

Monte-Carlo Tree Search (MCTS) is a search technique that has shown great success in playing games that can be represented as a sequence of decisions. In this paper we apply Nested Monte-Carlo Search, a single-player variant of MCTS, to the sequence of decisions made by the generating programs used by GödelTest, and show that this combination can efficiently generate random data structures which exhibit the specific properties that the test engineer requires. We compare the results to Boltzmann sampling, an analytical approach to generating random combinatorial data structures.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; E.1 [**Data**]: Data Structures; G.1.6 [**Numerical Analysis**]: Optimization; G.2.1 [**Discrete Mathematics**]: Combinatorics

Keywords

Software Testing; Data Structures; Search-Based Software Engineering; Nested Monte-Carlo Search

1. INTRODUCTION

The verification of some types of software requires test inputs in the form of complex data structures. Examples include information retrieval programs that traverse tree-like index structures, compilers that require syntactically correct source code to test functionality beyond the lexer and parser, and web applications that consume HTML and XML inputs. Often such structures are not bounded in terms of size, and so it is not possible to exhaustively enumerate all possible structures that could be supplied to the program; instead a finite set of structures may be sampled at random.

The question then arises as to the probability distribution from which these structures should be sampled. The test engineer may have knowledge as to the particular properties that are the most effective for the type of testing being performed: for example, certain properties may increase the likelihood of detecting a fault during system testing; while for performance testing, the size of the structure may be the property of interest. The challenge is to sample data structures with the particular properties desired by the engineer without introducing other unnecessary biases that could reduce diversity and thus the fault-detecting ability of the test data.

Techniques for sampling data structures at random include analytical approaches from the field of combinatorics, stochastic grammars, and non-deterministic generating programs. An example of the first of these techniques is Boltzmann samplers [5] which are efficient at uniformly sampling data structures of a desired mean size, but lack flexibility in terms of the type of data structure to which they can be applied. In our previous work [6], we introduced the GödelTest framework that uses non-deterministic generating programs and so is more flexible than Boltzmann samplers. We applied Differential Evolution to optimize the probability distributions associated with the non-deterministic constructs in the generating program so as to sample data structures with the desired properties. This approach was more effective than that of Boltzmann samplers, but still only a relatively small proportion of the generated data structures had properties that were very close or equal to the target values.

In this paper we propose a complementary approach to optimizing GödelTest generators with the objective of efficiently generating data structures that have *exactly* the required properties. By interpreting the generation of a data structure as a single-player game, we apply Nested Monte-Carlo Search (NMCS), a search technique that has been very effective in playing such games. We show that the efficiency of GT-NMCS (the combination of GödelTest and NMCS) can be equivalent to or better than Boltzmann samplers while retaining diversity in the sampled data structures, and can be applied to a wider range of data structures.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2576768.2598339.

In Section 2, we describe Boltzmann samplers, the GödelTest framework, and the Nested Monte-Carlo Search algorithm. We propose how NMCS can be applied to GödelTest in Section 3. Section 4 is an empirical investigation of the utility of the GT-NMCS algorithm. We discuss related work in Section 5, and summarize our conclusions and outline future work in Section 6.

2. BACKGROUND

2.1 Boltzmann Samplers

Boltzmann samplers, developed by Duchon et al. [5], permit the random generation of combinatorial data structures in a controlled and unbiased manner.

To create a sampler, the data structure is first expressed as grammar-like rules which describe the construction of the structure from simpler structures using a small set of operators. For example, general trees—that is, trees in which each node may have any number of children—can be expressed using the recursive rule:

$$\mathcal{G} = \mathcal{Z} \cdot \textbf{sequence}(\mathcal{G}) \quad (1)$$

where $\mathcal{G}$ represents the class of general trees, $\mathcal{Z}$ represents a single tree node, the operator $\cdot$ combines an instance of each of its two operands to form a new structure, and the operator **sequence** constructs a sequence of zero or more instances of its operand. Thus this rule specifies the construction of a general tree as a tree node (i.e. the root node) that has a sequence of zero or more subtrees that are themselves general trees.

Next, a probability distribution is assigned to each of the operators that require a choice to be made. For example, each time the operator **sequence** is encountered in the recursive rule above, a choice is made as to the number of subtrees that form the sequence. The probability distribution assigned to the **sequence** operator in a Boltzmann sampler is always a geometric distribution, but this distribution has a parameter, $0 < p \leq 1$, that controls its exact form: values of p closer to 1 favour shorter sequences.

The key feature of Boltzmann samplers is that it is relatively straightforward to calculate the parameters of the distribution associated with each operator that ensure the mean size of the generated structures has a chosen value. Moreover, when the parameters are set in this way, all structures with the same size, e.g. all general trees with 100 nodes, have the same probability of being emitted by the sampler: there is no unnecessary bias in the sampling process.

2.2 GödelTest

In the context of generating test data, Boltzmann samplers have three significant limitations. Firstly, the data structure must be expressible using the small set of operators that are supported by Boltzmann samplers; many data structures used in software, such source code or XML, cannot easily be expressed in this way. Secondly, Boltzmann samplers permit control only over the mean size of the sampled structures, but control over other properties—such as the height of a tree structure or the connectivity of a graph structure—may be necessary in order to efficiently detect faults in the software. Finally, few of the sampled structures necessarily have a size close to the mean value—the variance in the sizes can be large—and thus a sampler must often be supplemented by a potentially inefficient filtering mechanism to reject structures that are not of the desired size.

In earlier work, we developed the GödelTest framework to address these limitations [6]. GödelTest specifies the construction of the data structure using a non-deterministic program; the flexibility of such programs enables the generation of a much wider range of data structures than can Boltzmann samplers. The non-determinism may take the form of choices in the control flow of the generating program, e.g. whether or not to call a function; or choices as to the data processed, e.g. the sampling of a random integer.

```
function generator()
  node := create tree node
  add subtrees mult(generator()) to node
  return node
```

Figure 1: A GödelTest generator for general trees.

Figure 1 is an example of a GödelTest generator for general trees; the generator is described here using pseudocode[1]. In this example, **mult** is a GödelTest construct that returns an array consisting of zero or more objects created by calling the function specified as its argument (and is therefore the equivalent of the Boltzmann sampler **sequence** operator). This construct is an example of non-determinism—in this case, concerning the number of objects in the array—and so creates a *choice point*. GödelTest uses a *choice model* to resolve the non-determinism: each time a choice point is encountered during the execution of the generator program, the choice model is used to decide which choice to make. Internally, GödelTest labels each of the possible choices by a unique integer, and so the choice model supplies a single integer to identify which of the choices to make at the choice point. Figure 2 illustrates this process.

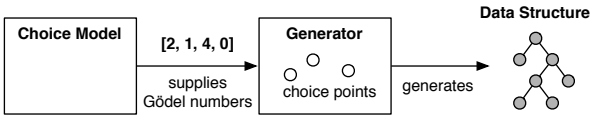


Figure 2: The GödelTest framework.

We refer to the integer supplied by the choice model as a *Gödel number*, and a single execution of a generator program is thus characterised by a sequence of Gödel numbers that we call a *Gödel sequence*. There is therefore a mapping from the set of Gödel sequences to the set of data structures that the program can generate. This mapping is a key feature of GödelTest: it enables the choice model to be abstracted from the specifics of the generating program.

In our previous work [6], the choice model was constructed in a similar manner to Boltzmann samplers: by assigning a local probability distribution to each choice point from which Gödel numbers were sampled. The parameters of the local probability distributions forming the choice model were optimized using Differential Evolution so that the generated data structures had, on average, the desired properties. We demonstrated that this a more flexible approach

[1]The GödelTest framework is not tied to a particular language: in our previous work the generating programs were written in Ruby, and for the empirical work of this paper the generators are implemented using the Julia language.

than Boltzmann samplers and thereby addressed the final two limitations described above. Firstly, the fitness metric used by the Differential Evolution (or other metaheuristic search algorithm) can be used to optimize the generator for any desirable property, not only the structure size. Secondly, by using probability distributions with more degrees of freedom, and permitting these distributions to be conditional on context information such as the recursion depth, a greater proportion of the sampled structures had properties that were closer to the desired values. Nevertheless, the proportion of sampled structures that had *exactly* the desired properties was still low. This is the motivation for the algorithm proposed in this paper: the construction of the Gödel sequence using Nested Monte-Carlo Search with the objective of satisfying the desired properties more precisely.

2.3 Nested Monte-Carlo Search

In this work we equate the generation of structured data using GödelTest to a single-player game, and use a variant of Monte-Carlo Tree Search (MCTS) that is particularly suited to this type of game: Nested Monte-Carlo Search (NMCS) proposed by Cazenave [3].

When used to play games, NMCS is applied to a *game tree*, the tree of potential game moves starting from the current game position in which nodes represent game states and edges valid moves. Figure 3 is an example of a game tree. The player has a choice of three moves: A, B, or C. If she makes move A, the next move she can make is either H, I, or J. If instead she makes move B, then no further moves are possible and the node represents a terminal state at which she has won or lost. The objective of NMCS is to determine a sequence of moves—a path in the game tree—that leads to a winning state.

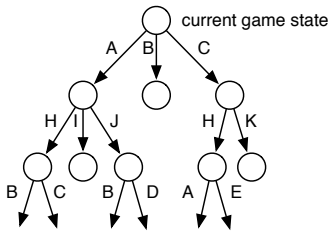


Figure 3: An example of the first three levels of a game tree. Edges are labelled by moves.

Many game-playing algorithms perform a search of the game tree that is as exhaustive as computational resources will permit. NMCS, and other variants of MTCS, take a different approach: the top-level moves are assessed by playing a relatively small number of game simulations through to conclusion. Each simulation starts with one of the top-level moves, may make a small number of deterministic moves to efficiently explore the first few levels of the game tree, but then makes all remaining moves at random until a terminal state is reached. These simulations provide an estimate of whether a given top-level move could lead to a winning state for the player. The estimate is noisy as a result of random nature of the simulation, but there is often a sufficient signal in the estimate to make an effective move choice.

At the current game state, NMCS performs one simulation of the game for each possible top-level move. This procedure is illustrated in Figure 4. At the terminal state reached by each simulation, a score is computed that is used to rank the terminal states and thus the associated top-level moves. The top-level move with the best simulation score is taken. The root of the game tree is now the state reached by taking this move, and the same procedure is applied in order to choose between top-level moves from this new root. The process repeats until no further moves are possible.

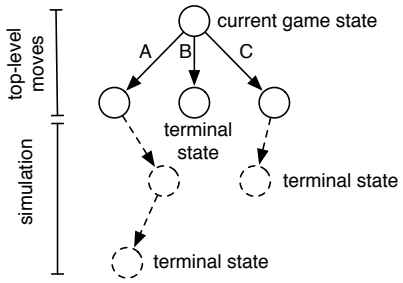


Figure 4: An illustration of Nested Monte-Carlo Search.

NMCS additionally tracks the best score, and the sequence of moves that gave rise to that score, over all the simulations performed so far. If none of the simulations at the current root state has a score greater than the tracked best score, the move from the sequence associated with the tracked best score is taken instead.

NMCS has one parameter: the nesting level which is an integer greater than or equal to 1. If the nesting level is 1, then the game simulations are random after making the first top-level move: at each step of the simulation, one of the possible moves is chosen at random, typically according to a uniform distribution. If the nesting level is greater than 1, then the game simulation itself uses NMCS—with a nesting level that is one lower—to choose moves. Figure 5 describes the NMCS algorithm in pseudocode.

3. APPLYING NMCS TO GÖDELTEST

3.1 GödelTest as a Game

Our proposal is that the generation of data structures using GödelTest can be equated to a single-player game, and that NMCS can be used to efficiently play the game to a 'winning' state that corresponds to a data structure with the desired properties. We refer to the combined algorithm as GT-NMCS.

Game states are partial Gödel sequences, and moves are the action of appending a single Gödel number to the end of the sequence. The root node is an empty Gödel sequence and a path through the game tree consists of moves that number-by-number build up a Gödel sequence. Figure 6 is an example of the first three levels of such a game tree.

To avoid a potential source of confusion, we clarify here that the game tree is *not* the same as the data structure that is being generated by GödelTest. In the empirical work of this paper, we select general trees as an example of a generated data structure since this enables a comparison with Boltzmann samplers. However GödelTest can be used to generate a wide range of data structures, and they need not be tree-like for the NMCS algorithm to be applied: a conceptual game tree exists for all GödelTest generating programs.

```
function nmcs(state, level)
  chosenSeq := empty sequence
  bestScore := -inf
  bestSeq := empty sequence
  while state is not terminal
    for each move possible from state
      state' := apply move to state
      if level == 1
        (score, seq) := random simulation from state'
      else
        (score, seq) := nmcs(state', level-1)
      end
    end
    highScore := highest score of the moves from state
    if highScore > bestScore
      bestScore := highScore
      chosenMove := move associated with highScore
      bestSeq := seq associated with highScore
    else
      chosenMove := first move in bestSeq
      bestSeq := remove first move from bestSeq
    end
    state := apply chosenMove to state
    chosenSeq := append chosenMove to chosenSeq
  end
  return (bestScore, chosenSeq)
```

Figure 5: The Nested Monte-Carlo Search algorithm, adapted from [3].

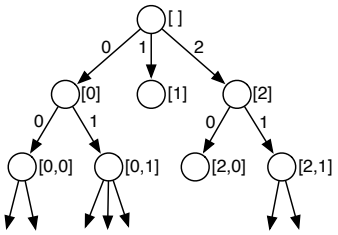


Figure 6: An example game tree for GödelTest. Edges are labelled by moves (Gödel numbers); nodes are labelled by game states (Gödel sequences).

3.2 Random Game Simulations

Normally a random game simulation in NMCS will choose moves according to a uniform distribution: each possible move has the same probability of being selected. However, this may not be the best strategy for some types of GödelTest generator: we hypothesise that the efficiency of GT-NMCS will be improved if the random simulation can incorporate domain knowledge that guides the simulation to data structures that are closer to having the desired properties. As an example, consider again the general tree generator of Figure 1: the **mult** construct is the equivalent of Boltzmann sampler **sequence** operator and so we argue that during game simulations, Gödel numbers for the **mult** choice point should be sampled from the same geometric distribution that would be used for the **sequence** operator of the general tree sampler of equation (1). Not all GödelTest generators will have an equivalent Boltzmann sampler that can be used for guidance, and so we explore the sensitivity of the GT-NMCS algorithm to the distributions used during game simulations as part of the empirical investigation of Section 4.

3.3 Choice Points with Infinite Support

Some GödelTest choice points have infinite support, i.e. there are an unlimited number of choices. The **mult** construct is an example of such a choice point: it returns an array consisting of zero or more objects and thus the associated Gödel number can be any non-negative integer. The NMCS algorithm can only assess a finite number of moves at each game state, and so we propose that at choice points with infinite support, only a small, finite subset of moves (Gödel numbers) are assessed, and this subset is chosen by random sampling from the same distribution that is used for the choice point during game simulation. The size of this finite subset becomes an additional parameter to the GT-NMCS algorithm, and we explore the effect of this parameter in the empirical investigation of Section 4.

3.4 Non-Terminating Games

Normally the GödelTest 'game' reaches a terminal state when the generating program halts, i.e. no further Gödel numbers are required in the sequence. It is possible, however, that an infeasibly long Gödel sequence may be required before reaching a terminal state or that the generating program may never halt. For example, a generator for general trees may attempt to construct trees of infinite size. For this reason, an upper limit is placed on the length of the Gödel sequence, the value of which will depend on the structures being generated. When this limit is reached no further moves are possible and thus the game state is a terminal state.

3.5 Score Function

The NMCS algorithm requires a score to be assigned to a terminal state in order to compare the state to other terminal states. In single-player games, the score is often expressed in terms of a reward of 1 if the terminal state is a winning state—which for GT-NMCS is a data structure that has exactly the properties we require—or 0 otherwise. However, we propose that the search algorithm will be more effective if the score is also able to quantify a non-winning terminal state in terms of how close the data structure's properties are to the target values.

For this reason, we use the following score function in the empirical work of this paper:

$$-\sum_{i=1}^{k} |p_i - t_i| \tag{2}$$

where the index i is over the k properties of interest, p_i is the value of the i^{th} property for the generated data structure, and t_i is the desired target value of this property. We do not claim that this is necessarily the most effective score function—it does not, for example, weight nor normalize the contribution from each property—but we suggest it is a reasonable metric to choose for the purposes of the empirical investigation.

4. EMPIRICAL INVESTIGATION

In the following four experiments, we investigate the performance and efficacy of the GT-NMCS algorithm and demonstrate its flexibility.

4.1 Expt A: Efficiency and Parameter Settings

This experiment evaluates the efficiency of the GT-NMCS algorithm in comparison to Boltzmann samplers, and ex-

plores the effect of parameter settings on the efficiency. We use a data structure that can be generated by both GödelTest and a Boltzmann sampler: the general tree example used earlier in this paper. Since Boltzmann samplers can be tuned only for size, we specify a target property of size 100 nodes.

The Boltzmann sampler for general trees is given by equation (1) in Section 2.1. By following the analysis of Duchon et al., we determine that the **sequence** operator in this sampler should use a geometric distribution with a parameter of 0.502512 in order to sample trees with a mean size of 100.

The GödelTest generator for general trees is listed in Figure 1. As proposed in Section 3.2, the probability distribution used for the **mult** choice point is same as that for the equivalent operator in the Boltzmann sampler: i.e. a geometric distribution with a parameter of 0.502512. The GT-NMCS algorithm has two parameters when applied to this generator: the nesting level used by NMCS, and the size of the finite sample of moves used for **mult** choice point. (We additionally set an upper limit on the Gödel sequence of 250 numbers in order to avoid infinite trees; this is much larger than a sequence of 100 numbers that would be required for a tree of size 100. For the same reason, this limit is also applied to the Boltzmann sampler.)

Our measure of efficiency is the number of trees that must be constructed in order to output a tree with the target size of 100. Each tree output by the Boltzmann samplers requires only one tree to be constructed internally, but few of the outputted trees will have exactly the target size. The GT-NMCS algorithm constructs multiple trees internally for each tree that is output—one for each game simulation performed—but our intention is that a higher proportion of the outputted trees have exactly the target size. We therefore count the number of game simulations per tree outputted (or equivalently, the number of score function evaluations) made by the GT-NMCS algorithm in order to compare its efficiency to that of Boltzmann samplers.

100 trees were sampled using the GT-NMCS algorithm at different combinations of the nesting level and move sample size parameters[2]. The results are shown in Table 1[3]. The column 'Tree Constructions' is the mean number of game simulations per sample, and the column 'Success Proportion' is the proportion of sampled trees which have exactly the target size of 100. The 'Efficiency' column is calculated by dividing the mean number of tree constructions by the success proportion to give an estimate of the mean number of tree constructions required to output a tree with the target size; lower values of this estimate indicate a more efficient algorithm.

There are no additional parameters to set for the Boltzmann sampler, but since only a low proportion of the generated trees have the target size, 100,000 trees were sampled in order to provide a more accurate estimate of the sampler's efficiency. The results are shown in the final row of Table 1.

[2]We include a move sample size of 1 which might be thought to be remove all choice from the NMCS algorithm since it creates a game tree in which there is only one choice of move at every node. However, NMCS also has the alternative of choosing the move from the best move sequence found so far by the search, and so—at least for some nodes—there are effectively two move choices.

[3]The data from the experiments is available at: `http://www.cs.york.ac.uk/~smp/supplemental`.

Table 1: Expt A – Algorithm Efficiency

Nesting Level	Sample Size	Tree Constr'ns	Success Proport'n	Efficiency
1	1	32.91	23%	143.1
1	2	136.7	60%	227.8
1	4	380.3	94%	404.6
1	8	792.1	99%	800.1
1	16	1,600	100%	1,600
2	1	1,921	41%	4,684
2	2	13,620	71%	19,180
2	4	72,890	94%	77,540
3	1	60,770	41%	148,200
Boltzmann Sampler		1	0.032%	3,125

The number of tree constructions increases quickly with nesting level, but there is little increase in the success proportion for this particular generator. We speculate that a higher nesting level may be required for more complex generating programs, but for this general tree generator, a nesting level of 1 provides the best efficiency. Within a nesting level, the efficiency decreases as the move sample size increases, but we note that for all sample sizes between 1 and 16 at a nesting level of 1, the efficiency is better than that of the equivalent Boltzmann sampler.

The efficiency is assessed in terms of the number of tree constructions rather than the 'wall clock' run time in order to minimize threats to validity arising from the implementation of the respective algorithms and the noise of other processes running on the same server. With these caveats in mind, we are neverthless able to give an indication that the better efficiency of the NMCS algorithm is also reflected in the run time, although only at the lower move sample sizes. The Boltzmann sampler takes on average 0.144s to generate a tree of size 100; the NMCS algorithm takes 0.0673s, 0.0713s, 0.122s, 0.241s, and 0.482s for sample sizes of 1, 2, 4, 8, and 16 respectively at nesting level 1 on the same hardware.

4.2 Expt B: Sampling Uniformity

This experiment measures the extent to which the algorithm samples uniformly from the space of data structures with the desired properties, and how sensitive this is to the move sample size parameter. We again use general trees of size 100 as target data structure.

Many different characteristics could be used to assess how the sampled trees fill the space of all general trees of size 100. However, a particular concern is that because the GödelTest generator constructs the tree in a depth-first pre-order traversal that moves from left to right, the NMCS algorithm may introduce a bias that skews the trees to the left or to the right. Moreover, it is possible that all the trees generated by GT-NMCS are identical or very similar.

We therefore measure a characteristic of the generated trees that we term 'median coordinate', calculated using the following recursive procedure. Each node is passed an interval of the real numbers by its parent (the root node starts with the unit interval $[0, 1]$). The node is labelled with a coordinate that is the midpoint of this interval. The interval is then partitioned into subintervals of equal length, one for each of the node's children. The subintervals are then passed to the child nodes, with the interval closest to 0 being passed to the leftmost child and so on. After this procedure, nodes on the left of the tree will be labelled with coordinates

closer to 0, and those on the right with coordinates closer to 1. An example is shown in Figure 7. The median coordinate of all the nodes is calculated; a median other than 0.5 indicates a skew to the left or right for an individual tree. For the example tree of Figure 7, the median coordinate is 0.375 and indeed the tree is skewed to the left.

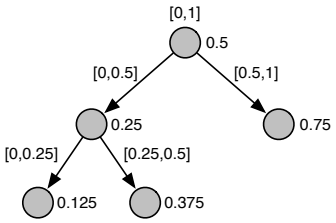


Figure 7: A tree in which the nodes are labelled by coordinate values. The intervals passed from parent to child nodes are shown on the edges.

It is not a problem if individual trees possess a skew; indeed, in a diverse sample of trees we would expect a range of skews. Instead our concern is whether the distribution of skews (measured by median coordinate) across the sample of trees generated by NMCS is consistent with uniform sampling from the space of trees of size 100. This would indicate that the the NMCS algorithm does not introduce a bias in terms of the skew *and* that it generates a diverse range of trees. We do not calculate the ideal distribution of median coordinates directly, but derive it empirically from the Boltzmann sampler which, by design, samples trees of the same size uniformly (see Section 2.1).

The empirical procedure was to generate trees of size 100 (filtering out trees of other sizes) from the Boltzmann sampler until a sample of 200 such trees was obtained. The median coordinate was calculated for each tree in the sample. The same procedure was applied to the GT-NMCS algorithm using a nesting level of 1, and move sample sizes of 1, 2, 4, 8, and 16.

The distributions of the median coordinates of the trees generated by each of the algorithms are summarised by the boxplots of Figure 8. It can be seen that the GT-NMCS algorithm using a sample size of 1 has a bias in that trees tend to be skewed to the right, but the trees generated at higher sample sizes have a distribution that is visually similar to that of the Boltzmann sampler. (The boxplot for the Boltzmann sampler suggests a small skew to the left, but we suspect this a chance occurrence as result of the finite sample size of 200 trees.)

We may quantify the similarity between the distributions using a two-sample Kolmogorov-Smirnov test; the null hypothesis is that the two samples are drawn from the same continuous probability distribution. Such a test between the Boltzmann sampler and GT-NMCS with move sample size 1 gives a p-value that is much less than 1%, indicating a statistically significant difference in the distributions. Between the Boltzmann sampler and other GT-NMCS sample sizes, the test gives p-values that are all greater than 25%. This result provides evidence that, for this generator, GT-NMCS using sample sizes of 2 and above samples trees close to uniformly from the space of trees with target size 100. We note, however, that this analysis considers only one characteristic of this space of trees.

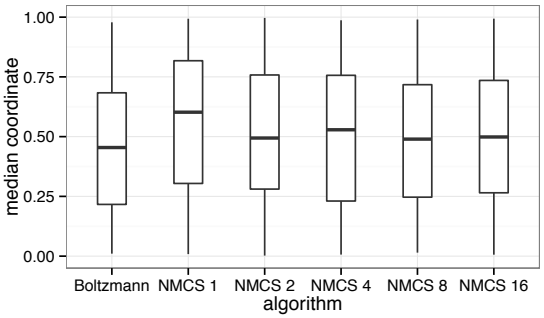


Figure 8: Boxplots summarizing the median coordinate distributions for each of the algorithms. (The number after the NMCS label indicates the move sample size.)

4.3 Expt C: Choice Point Distributions

This experiment explores the sensitivity of the algorithm to the probability distributions from which moves are sampled during game simulations.

The empirical procedure was to repeat experiments A and B for sample sizes between 1 and 16 at a nesting level of 1, while additionally varying the distribution used at the **mult** choice point in the general tree generator. The distributions investigated were geometric distributions with parameters 0.3, 0.4, 0.6, and 0.7; and a discrete uniform distribution over the integers 0 to 3.

The efficiency results are shown in Table 2 (∞ indicates that none of the 100 sampled trees had the target size of 100), and the uniformity results—as p values of Kolmogorov-Smirnov test—in Table 3. In both tables, the row for the geometric distribution with a parameter of 0.502512 (that derived using Boltzmann sampler analysis) summarises the results of experiments A and B and is included for comparison.

The results show that the efficiency of the algorithm is sensitive to the distribution used at the choice point: the efficiency is worse the further the distribution is from that derived using Boltzmann sampler analysis. However, the sensitivity of algorithm efficiency to the choice of distribution is ameliorated by larger sample sizes: at sizes 8 and 16 there is very little difference in efficiency between the Boltzmann sampler distribution and the geometric distributions with parameters 0.6 and 0.7.

The sampling uniformity is also sensitive to the distribution: relatively small deviations result in distributions of the median coordinate that are significantly different from that of the Boltzmann sampler (p-values less than 5% in table 3). There is again evidence that a higher sample size can ameliorate this sensitivity, but only for the geometric distribution with a parameter of 0.6.

4.4 Expt D: Flexibility

In this final experiment, we demonstrate the flexibility of the GT-NMCS algorithm by generating red-black trees, a data structure that cannot be expressed in the grammar of Boltzmann samplers, and by specifying more than one target property.

Red-black trees, described by Guibus and Sedgewick [8], are a form of binary tree often used in software to organise

Table 2: Expt C – Algorithm Efficiency

Distribution	Sample Size 1	2	4	8	16
geom(0.3)	∞	∞	∞	3,790	2,035
geom(0.4)	2,475	576.2	561.4	891.9	1,691
geom(0.502512)	143.1	227.8	404.6	800.1	1,600
geom(0.6)	136.1	202.6	401.1	800.1	1,600
geom(0.7)	482.0	301.7	413.8	800.6	1,600
uniform(0,3)	∞	∞	∞	1,867	1,885

Table 3: Expt C – Kolmogorov-Smirnov p-value

Distribution	Sample Size 1	2	4	8	16
geom(0.3)	N/A	N/A	N/A	0.0%	0.0%
geom(0.4)	0.0%	0.0%	0.0%	0.0%	0.0%
geom(0.502512)	0.0%	27.0%	39.3%	71.1%	32.7%
geom(0.6)	0.0%	0.0%	0.0%	6.8%	5.2%
geom(0.7)	0.0%	0.0%	0.0%	0.0%	0.0%
uniform(0,3)	N/A	N/A	N/A	0.0%	0.0%

data. They are guaranteed, by construction, to be nearly balanced—every leaf node has almost the same depth—and thereby permit efficient retrieval of the data. Each node in the tree is annotated either red or black, and the colour annotations must conform to four constraints. The first three are relatively straightforward to implement in either a grammar-based or non-deterministic program generator: the root node, the children of a red node, and leaf nodes must all be black. The fourth constraint is a property of the tree as a whole: the number of black nodes on the path from the root to a leaf node must be the same for all leaves. Enforcing this fourth constraint (without explicitly specifying the number of black nodes on the path) requires state information to be stored and retrieved during the generation process. This ability is not available in a Boltzmann sampler nor many grammar-based generators, but is possible using a non-deterministic generating program such as GödelTest.

We demonstrate this flexibility using the GödelTest generator for red-black trees shown in Figure 9 that enforces all four constraints. The argument `pcolour` is the colour of this subtree's parent node, and the argument `pbdepth` is the number of black nodes on the path from the root to this subtree's parent. The global variable `gbdepth` stores the number of black nodes that must be on the path from the root to each leaf node; this value is set by the first leaf node generated, and is -1 until then. The GödelTest **maybe** construct returns either its argument or nothing, and so the expression **maybe**('a')=='a' evaluates to either true or false. There is a choice of two Gödel numbers at a **maybe** choice point—0 and 1—and a Bernoulli distribution (a biased coin toss) is used during game simulation.

We demonstrate that GT-NMCS can generate data structures that simultaneously satisfy multiple property targets by specifying a tree size of 101 nodes *and* a tree height of 8. (Each non-leaf node in a binary tree has 2 children, and so the total number of nodes—including the root node—is odd. Since a red-black tree is nearly balanced, the possible heights are constrained to a limited range: a height of 8 is near the upper boundary of this range for tree of size 101.)

The choice point associated with the **maybe** construct has finite support and so we do not need to take a sample of moves; instead both the moves of 0 and 1 are always considered by the NMCS algorithm. During a random game simulation, a parameter of 0.5 is used for Bernoulli distribution associated with the **maybe** construct. We specify a nesting level of 2 in order to demonstrate the ability to generate a high proportion of the data structures with the desired properties; a nesting level of 1 has better efficiency for this particular generator but fewer of the generated structures would have exactly the desired properties.

```
global gbdepth=-1
function generator(pcolour:=red, pbdepth:=0)
  node := create tree node
  if pcolour!=red and maybe('a')=='a'
    colour node red
    bdepth := pbdepth
  else
    colour node black
    bdepth := pbdepth+1
  end
  if (gbdepth==-1 and (node is red or maybe('a')=='a'))
  or (bdepth < gbdepth)
    add subtree generator(node colour, bdepth) to node
    add subtree generator(node colour, bdepth) to node
  else if gbdepth==-1
    gbdepth := bdepth
  end
  return node
```

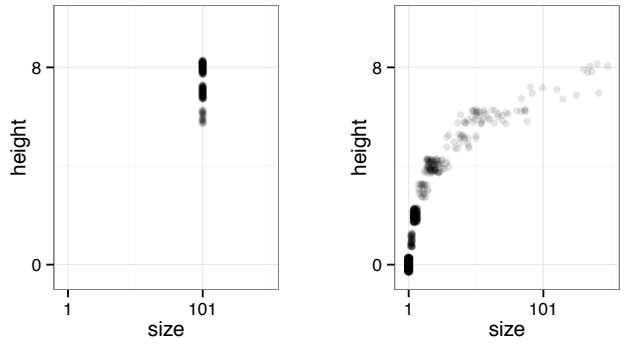
Figure 9: GödelTest generator for red-black trees.

Figure 10(a) shows the size and height of a sample of 1,000 trees generated by the GT-NMCS algorithm. (Since many trees have the same combination of size and height, a small amount of jitter is added, and the points are semi-transparent, in order to clarify the distribution.) As can be seen, all the generated trees have the desired size, and a large proportion have the desired height. Further analysis shows that 50.4% of the generated red-black trees have *both* the target size of 101 *and* the target height of 8.

We cannot generate red-black trees using a Boltzmann sampler as a comparison. Instead we use a 'random sampler': the generator of Figure 9 in which choices at the **maybe** constructs are made by random sampling from a Bernoulli distribution with a 'default' parameter of 0.5 rather than being guided by NMCS. Figure 10(b) shows the distribution of 1,000 red-black trees generated in this way. It can be seen the random sampler would generate far fewer trees of the target size and height than the GT-NMCS algorithm.

5. RELATED WORK

A number of test data generation techniques utilize non-deterministic programs of a type similar to that used by GödelTest. For example, UDITA [7] is a sophisticated Java-based framework that generates sets of test cases by exhaustively enumerating all points of non-determinism in the generating program up to a chosen bound. QuickCheck [4] is a framework for testing programs written in Haskell and a number of other languages. It facilitates the creation of custom generating programs that, in constrast to UDITA, sample test data stochastically rather than exhaustively. The generating programs of QuickCheck are therefore similar in objective to those used by GödelTest, but a key difference is that the probability distributions used by the QuickCheck generators must be optimized manually to generate test data with desirable properties, while GödelTest can perform this optimization automatically using metaheuristic search.



(a) (b)

Figure 10: The size and height of 1,000 red-black trees generated by (a) the GT-NMCS algorithm; (b) a random sampler.

Similarly, a wide range of test data generation techniques utilize stochastic grammars. The algorithms described by Beyene and Andrews [1], and by Poulding et al. [9], are particularly notable in that they optimize the grammar using metaheuristic search in order to achieve coverage of the software-under-test. These grammar-based generators are inherently less flexible than non-deterministic generating programs such as GödelTest, but are potentially easier to optimize in order to satisfy target properties.

The GödelTest abstraction of choices as an integer sequence has parallels in Grammatical Evolution (GE) [10]. GE interprets an integer sequence as production rule choices in a grammar and applies evolutionary algorithms to optimize the sequence. However, GE constructs and interprets the sequences in a different way to the GT-NMCS algorithm used in this paper. The GE sequence is normally fixed-length (the sequence is re-used should more integers be required), and the integer values are not specific to the use to which they are put (the modulus is taken base the number of choices). GE optimizes the sequence as a whole, while GT-NMCS constructs an optimal sequence integer-by-integer.

NMCS and other MCTS variants have achieved great success in playing games and are often competitive with human experts [2]. We suspect that they will prove to be effective search methods for software engineering problems that can be expressed as a game tree, but as far as we are aware, this paper is the first use of such algorithms in search-based software engineering.

6. CONCLUSIONS

In this paper we proposed the application of Nested Monte-Carlo Search to our previously described GödelTest framework. We have demonstrated that for general trees, the GT-NMCS algorithm has an efficiency that is better than the equivalent Boltzmann sampler, and—with appropriate parameter settings—samples structures uniformly and so maintains diversity in the test data. However, both performance and uniform sampling are sensitive to the default distributions used during game simulations. We also demonstrated the flexibility of the framework with regards of both the type of data structure to which it can be applied and the properties of those structures that can be targeted.

As future work, we will explore the performance of the GT-NMCS algorithm on more complex data structures that require generating programs with many choice points, and investigate how the sensitivity of the algorithm to the choice of distributions during game simulation can be reduced. In addition, we would like to identify the characteristics of generating programs that determine which GT variant—GT-NMCS or GT using Differential Evolution—is the most effective for that generator.

7. ACKNOWLEDGMENTS

This work was funded by EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering; and by The Knowledge Foundation (KKS) through the project 20130085, Testing of Critical System Characteristics (TOCSYC). The authors would like to thank Ed Powley for his feedback on a draft on this paper.

8. REFERENCES

[1] M. Beyene and J. Andrews. Generating string test data for code coverage. In *Proc. IEEE Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 270–279, 2012.

[2] 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*, 4(1):1–43, 2012.

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

[4] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proc. ACM SIGPLAN Int'l Conf. Functional Programming (ICFP)*, pages 268–279, 2000.

[5] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.

[6] R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *Proc. IEEE Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 350–359, 2013.

[7] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proc. ACM/IEEE Int'l Conf. Software Engineering (ICSE)*, volume 1, pages 225–234. ACM, 2010.

[8] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. Annual Symp. Foundations of Computer Science*, pages 8–21, 1978.

[9] S. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley. The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing. In *Proc. Genetic and Evolutionary Computation Conf. (GECCO)*, pages 1477–1484, 2013.

[10] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution : Evolving programs for an arbitrary language. In *Proc. European Workshop on Genetic Programming (EuroGP)*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96, 1998.