Heuristic Model Checking using a Monte-Carlo Tree Search Algorithm

Simon Poulding
Software Engineering Research Laboratory
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
simon.poulding@bth.se

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

ABSTRACT

Monte-Carlo Tree Search algorithms have proven extremely effective at playing games that were once thought to be difficult for AI techniques owing to the very large number of possible game states. The key feature of these algorithms is that rather than exhaustively searching game states, the algorithm navigates the tree using information returned from a relatively small number of random game simulations. A practical limitation of software model checking is the very large number of states that a model can take. Motivated by an analogy between exploring game states and checking model states, we propose that Monte-Carlo Tree Search algorithms might also be applied in this domain to efficiently navigate the model state space with the objective of finding counterexamples which correspond to potential software faults. We describe such an approach based on Nested Monte-Carlo Search—a tree search algorithm applicable to single player games—and compare its efficiency to traditional heuristic search algorithms when using Java PathFinder to locate deadlocks in 12 Java programs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Model Checking; G.1.6 [**Numerical Analysis**]: Optimization; E.1 [**Data**]: Data Structures; I.6.8 [**Simulation and Modelling**]: Monte-Carlo

Keywords

Model Checking; Monte-Carlo Tree Search; Search-Based Software Engineering

1. INTRODUCTION

Model checking represents software as an abstract model, and explores the states of this model to verify properties of the software [4]. However there may be too many states, particularly in concurrent software, to feasibly check all of the states; a more realistic objective may be to efficiently locate counterexamples—states within the model at which desirable properties are violated—using a heuristic to guide the search.

In this paper we propose the use of a Monte-Carlo Tree Search (MCTS) algorithm called Nested Monte-Carlo Search (NMCS) [3] as a heuristic search method for model checking. Our motivation is that MCTS algorithms have shown great success in playing games, such as Go, which are thought to be particularly difficult problems for AI techniques to solve [2]. Our objective is to improve the time efficiency of heuristic search in model checking, i.e. the time required to find a counterexample state.

We propose an extension to the NMCS algorithm when it used for heuristic model checking: a parameter controlling the length of the random simulations it uses to choose which state to move to next. Our rationale is that the path to a terminal state in the model may be so long that performing simulation until such a state is reached could lead to a computationally inefficient algorithm. The effect of the parameter is investigated by locating deadlocks in 12 Java programs from the Software-artifact Infrastructure Repository, and comparing the time and space efficiency of NMCS to the out-of-box heuristic search algorithms that ship with Java PathFinder (JPF). The results provide compelling evidence of the potentially superior efficiency of NMCS when the length of the random simulation is set appropriately.

Section 2 provides an overview of model checking using heuristic search and describes the standard NMCS algorithm. In Section 3, we propose how NMCS could be used for model checking and identify the need for the parameter controlling the length of random simulations. The proposed algorithm is investigated empirically in Section 4. We relate this paper to existing work in Section 5, and summarise our conclusions in Section 6.

2. BACKGROUND

2.1 Model Checking

Model checking is a technique for verifying properties of the software. Often it is a static technique that builds an abstract model of the software, although Java PathFinder (JPF), the model checker used in the empirical investigation of this paper, builds the model by dynamically executing the software. The model is typically a directed graph, where each node represents a state of the software and edges represent possible transitions between the states.

A state is characterised by the current point of execution and the values of the program's data. When the software is

GECCO '15, July 11 - 15, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739480.2754767

concurrent, the state is additionally characterised by which thread is currently active. One of the particular advantages of model checking as a verification technique for concurrent software is that it can consider all possible interleaving of threads in contrast to dynamic testing during which there may be no method to control thread interleavings.

Model checking can be used to verify that desirable properties hold for every state of the software; for example, the absence of deadlocks and race conditions in concurrent software. However the number of states in the model may be so large that such an exhaustive check of each state is infeasible. Instead the approach taken may instead be to efficiently locate counterexamples: states at which the desirable property is violated.

2.2 Heuristic Search

One approach for finding counterexamples efficiently, i.e. by visiting as few states as possible, is to use a suitable heuristic to guide the exploration of states in the model [7]. A heuristic in this context is a function that maps a state to a numeric value that quantifies how close a state is to violating the desired property and so being a counterexample. This has obvious similarities to the fitness function used in metaheuristic techniques and, indeed, in this paper, we interpret the exploration of a model for counterexamples as sharing many characteristics with metaheuristic search on a fitness landscape.

Best-first search is an example of heuristic search. Starting from a root state, each of the state's children is considered and the heuristic calculated for each. These child states are added to a queue. The state with the best heuristic value is then retrieved from the queue and similarly explored: each of its child states are considered and the heuristic is calculated. The search proceeds in this manner until a counterexample is found or no more states can be reached.

2.3 Nested Monte-Carlo Search

Monte-Carlo Tree Search (MCTS) methods are machine learning algorithms that derive a path through a tree, where each node represents a decision to be made. They have demonstrated significant success in the domain of automated game playing, and in this context, the tree is a game tree in which nodes represent game states, and the decision to be made is which action to take at each turn in order to improve the computer's ability to win the game. Browne et al. provide a detailed survey of the current state-of-the-art in MCTS and, in particular, the application of these methods to game playing [2].

The key feature of MCTS methods are that at each node in the tree, the possible decisions are assessed by making a small number of random simulations of future game play from the current state to a state at which the game ends. The premise is that although a small sample of random simulation will provide a noisy signal as to which move to take, there is nonetheless sufficient signal that over a series of moves the computer's game play is effective, and the use of a small set of simulations is much more efficient than an exhaustive exploration of the game tree.

In this work we propose the use of Nested Monte-Carlo Search (NMCS), a variant of MCTS described by Cazenave for the purpose of playing single-player games [3]. To decide which move to take at the current game state, NMCS makes one simulation for each of the possible moves. The simulation starts by making that move and then proceeds by random moves until the simulated game comes to an end, at which point the end state is assessed by a reward function. The simulation that returns the best reward determines the move that is taken, or, if no simulation is better than the simulation that led to the current game state, the move is determined by that earlier simulation.

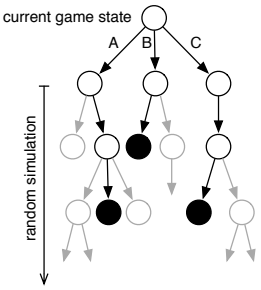

Figure 1: NMCS applied to a game tree

Figure 1 illustrates the algorithm for a game state with three possible moves, A, B, and C. A random simulation (represented by the black arrows) is run from each of the moves until an end state (indicated by a filled node) is reached where no further moves are possible. The reward function is measured at the end states, and whichever of A, B, and C leads to the end state with the best reward (say, C) is the move that is taken. The algorithm is then applied again at the new node reached when the move C is taken.

Instead of random simulation, the algorithm may instead use NMCS itself to determine the moves to make during simulation. This is the nested aspect of the algorithm, and a parameter—the nesting level taking integer settings of 1 or greater—controls the degree of nesting.

3. PROPOSED ALGORITHM

Our proposal is to apply a form of NMCS as heuristic search method in model checking. Our hypothesis is that we may realise, for model checking, the same efficacy and efficiency of Monte-Carlo Tree Search methods that is exhibited in the context of automated game play.

The transfer of NMCS to the domain of model checking is, for the most part, straightforward: states in the model are equivalent to states in the game tree, and the model-checking heuristic function is equivalent to the game-playing reward function. However, a complication arises with the stopping criterion for the simulations. In game playing, the simulation continues until no further moves are possible (one of the players has won, or a stalemate has been reached). In model checking, following a path of model states until an end node is reached may not be practical. Not only can the model be extremely large, but is often a graph rather than a tree: many cycles in the graph may require an extremely long path if an end state is to be reached.

For this reason, we introduce a new parameter to the form of NMCS used in this work: the *simulation depth*. This parameter limits the length of the path that is followed during a simulation. The heuristic is measured at the state when this limit is reached (or at an end state if one is reached before the limit). Figure 2 describes the proposed NMCS algorithm using pseudo-code.

```
function nmcs(state, nestingLevel, simDepth,
                  currentLevel:=nestingLevel)
  chosenPath := empty sequence
  bestValue := -infinity
  bestPath := empty sequence
  depth = 0
  while state is not terminal and
    (currentLevel == nestingLevel or depth < simDepth)
    for each move possible from state
      state' := apply move to state
      if currentLevel == 1
        (value, path) := simulate(state', simDepth)
      else
        (value, path) := nmcs(state', nestingLevel,
                         simDepth, currentLevel-1)
      end
    end
    highValue := highest value of the moves from state
    if highValue > bestValue
      bestValue := highValue
      chosenMove := move associated with highValue
      bestPath := path associated with highValue
    else
      chosenMove := first move in bestPath
      bestPath := remove first move from bestPath
    end
    state := apply chosenMove to state
    depth := depth + 1
    chosenPath := append chosenMove to chosenPath
  end
  return (bestValue, chosenPath)

function simulate(state, simDepth)
  chosenPath := empty sequence
  depth = 0
  while state is not terminal and depth < simDepth
    pick chosenMove from state at random
    state := apply chosenMove to state
    depth := depth + 1
    chosenPath := append chosenMove to chosenPath
  end
  value := heuristic(state)
  return (value, chosenPath)
```

Figure 2: The Nested Monte-Carlo Search algorithm, adapted from [3], and extended with a limit on simulation depth

4. EMPIRICAL INVESTIGATION

4.1 Research Questions

The objective of the empirical investigation is to assess the efficiency of our proposed NMCS algorithm for model checking in comparison to the other heuristic algorithms. We use Java PathFinder (JPF) as the model checker for these experiments, and for this reason the comparison are against the best-first search and beam search algorithms supplied with JPF.

We consider three aspects of efficiency. Firstly, the proportion of algorithm trials—the success rate—during which a counterexample state is found. Secondly, the time taken to find a counterexample. The time is measured in two ways: (a) the number of states visited by the algorithm, and, (b) the elapsed wall-clock time. The difference between the two is that the number of visited states is independent of the performance of algorithm's implementation. Thirdly, the space required by an algorithm trial, measured as the maximum memory consumption reported by JPF.

The research questions we seek to answer are:

RQ1 To what extent does the efficiency of NMCS depend on the setting of the simulation depth parameter?

RQ2 How does efficiency of NMCS compare to JPF's best-first and beam search algorithms?

4.2 Java PathFinder

Java PathFinder (JPF) differs from most model checking tools in that it dynamically executes Java bytecode rather than an abstract model of the software [12]. JPF is, in essence, a Java virtual machine (itself written in Java) and this enables JPF to control and measure many aspects of software's execution. In particular, it can enumerate and control the choices made at non-deterministic points in the program, such as the generation of pseudo-random numbers and—most importantly for the experiments here—the interleaving of separate threads in concurrent program. The degree control of JPF exerts over the interleaving of threads is not available when using a standard Java virtual machine, and thus JPF is particularly useful in finding potential faults in concurrent programs.

JPF ships with a number of search algorithms, such as depth-first search, breadth-first search, and heuristic search. The default heuristic search algoritm, implemented as the `SimplePriorityHeuristic` class, is a form of breadth-first search of the type described above in Section 2.2. A priority queue (which, by default, has a maximum size of 1024 entries) is maintained of all the states visited so far, and the state with the best heuristic is explored next by calculating the heuristic value at each of its child state. These states are added to the priority queue, and then again the state with the best heuristic is explored; this need not be one of the children of the state just explored. The exploration process continues until there are no further states to explore.

This search can be modified, using the parameter setting `search.heuristic.beam_search=true`, to a form of beam search. The effect of this setting is to clear the priority queue each time a new state is chosen to explore, and this guarantees that the next state explored *is* one of the children of the state just explored. The exploration process continues until the current state has no children, which indicates either the end of program, or a deadlock condition.

JPF is highly extensible through mechanisms such as class inheritance and specifying call-back listeners when specific event occurs during the execution of the program. It is the former mechanism we use to implement Nested-Monte Carlo Search in the form of a new class `NMCSSearch` that inherits from the base `Search` class.

4.3 Software-Under-Test

In the following experiments the counterexample that is sought is a deadlock, i.e. a state in which no thread is able to proceed since it is waiting on one or more other threads.

The SUTs are taken from the Software-artifact Infrastructure Repository at the University of Nebraska-Lincoln [5]. The repository includes a number of Java SUTs with seeded concurrency faults and that may be run using JPF. For the experiments we select those SUTs from the repository where the only seeded concurrency fault is a deadlock.

We exclude one of the SUTs, `pool5`, since it raises a `NoSuchElementException` exception rather than a deadlock in our execution environment. We exclude a further SUT,

`piper`, because it consistently exhausts the memory when using best-first search, and so would limit the quantitative comparisons that would be possible against the remaining algorithms. This leaves 12 SUTs, the names of which are listed in the first column of table 2. All the SUTs were executed with the input arguments used in the JPF scripts provided by the repository.

4.4 Heuristic

The 'most blocked' heuristic is used to guide the search for deadlocks, and is calculated as:

$$h_{\text{mostblocked}} = N_{\text{alive}} - N_{\text{runnable}} \quad (1)$$

where N_{alive} and N_{runnable} are the numbers of threads that have not finished ('alive') and that are currently available to execute ('runnable'), respectively. The higher the value of $h_{\text{mostblocked}}$, the more threads are prevented from executing ('blocked'), and thus the closer the state is to being a deadlock.

We choose this heuristic since it is used commonly in model checking for this purpose, and because JPF includes it as an example heuristic. For NMCS we utilise the same code to calculate this heuristic as does the 'out-of-the-box' JPF implementation in the class `MostBlocked`.

4.5 Method

The search algorithms evaluated in the experiment are:

- The JPF implementation of best-first search. This algorithm is labelled 'B-F' in the results below.
- The JPF implementation described as beam search. This algorithm is labelled 'Beam' in the results.
- Our proposed NMCS algorithm. Since this algorithm has two parameters—the nesting level and simulation depth—we considered 10 different parameter combination. All had the nesting level set to 1, and the simulation depth set to powers of 2 between 1 and 512. (We have experimented with different nesting levels, but for clarity and brevity, we report here only the experiments at nesting level 1 as algorithm with this parameter setting typically demonstrated the best efficiency.) These ten algorithm variants are labelled 'N1,*d*', where *d* is the simulation depth, in the results.

For each combination of SUT and search algorithm, 30 trials of the algorithm were run. Since all the search algorithms considered here are stochastic, each trial used a different sequence of pseudo-random numbers by means of the JPF parameter setting `cg.randomize_choices=VAR_SEED`. These 30 trials will therefore provide a more accurate estimate of each algorithm's *average* efficiency.

For each trial, the following responses were recorded from statistics provided by JPF:

- Whether or not a deadlock state was found.
- The number of states visited. We calculate this as the sum of the JPF counts of 'new', 'visited', and 'end' states. Therefore it is not necessarily the number of unique states visited, but instead is the number of transitions to new states made by JPF that require a decision by the search process. This value is used as a metric because it is indicative of how much of the search space that algorithm has had to explore to find a counterexample, and in JPF transitioning to states is a computationally expensive action.
- The elapsed (wall-clock) time taken (JPF records this to the nearest second).
- the maximum amount of memory consumed by JPF.

The experiments were run on a mid-2013 MacBook Air 13" which had a 1.3GHz Intel Core i5 CPU, and 8 GB of 1600 MHz DDR3 memory. The operating system was OS X 10.10.2. Java PathFinder was version 7.0 (rev 1225), and the JRE was Oracle Java 1.7.0_40.

4.6 Results and Analysis

4.6.1 Success Rate

We define success rate as the proportion of the JPF trials that found a deadlock state. In table 1, we present this result for each combination of algorithm and SUT: the figure is therefore the proportion of the 30 trials for each algorithm / SUT combination that was successful (rounded to 3 significant figures).

The final row, labelled 'Average (Arith. Mean)', is average success rate of each algorithm considered over all 12 SUTs, and is calculated using the arithmetic mean.

These results are consistent with the nature of the algorithms. Best-first search is an exhaustive algorithm: each state is explored until a deadlock is found or there are no more states to explore. Thus for these SUTs which have seeded deadlocks, all trials would be expected to be successful. In contrast, both Beam Search and NMCS take a single path to an end state, the choice of nodes on the path being informed by the heuristic. The end state need not be a deadlock—it might also be the end of the program—and thus there is not reason to expect a success rate of 1 for these two algorithms.

We make no further comment here on these results since the success rate is not an efficiency metric that we consider in isolation. Instead we use it to adjust the time efficiency metrics so that we can compare the algorithms in a consistent manner; this process is described in the following section.

4.6.2 Time Efficiency

In practice, a tester would run repeated trials of the algorithm until a deadlock were found. In order to compare algorithms in terms of the time efficiency, we therefore calculate the average *total* time required to a deadlock, i.e. across the multiple trials of the algorithm that the tester may need to perform. We calculate this total time by combining the raw time data with the success rates shown in table 1 as follows.

If the success rate is ρ, then the average *total* time taken, i.e. over multiple failed and one successful trial, to find the deadlock, $\bar{t}_*$, is:

$$\bar{t}_* = \frac{(1-\rho)}{\rho}\bar{t}_- + \bar{t}_+ \quad (2)$$

where $\bar{t}_+$ and $\bar{t}_-$ are the average times of failed and successful trials respectively. The factor $1-\rho/\rho$ arises since this process is a sequence of independent Bernoulli trials and so the number of failed trials until a success has a Geometric distribution with its mean given by this factor.

The average time of a successful trial, $\bar{t}_+$ is calculated as:

$$\bar{t}_+ = \frac{1}{\rho M} \sum_{i \in I_+} t_i \tag{3}$$

where t_i are times observed for the M trials—both failed and successful—and I_+ is the set of indices, i, for successful trials. The denominator is ρM since, from the definition of ρ this must be the number of successful trial and thus the size of the set I_+. (Here, $M = 30$ for each algorithm / SUT combination.)

Similarly, $\bar{t}_-$ is calculated as:

$$\bar{t}_- = \frac{1}{(1-\rho) M} \sum_{i \in I_-} t_i \tag{4}$$

Substituting for $\bar{t}_+$ and $\bar{t}_-$ in (2) gives:

$$\begin{aligned} \bar{t}_* &= \frac{(1-\rho)}{\rho} \cdot \frac{1}{(1-\rho)M} \sum_{i \in I_-} t_i + \frac{1}{\rho M} \sum_{i \in I_+} t_i \\ &= \frac{1}{\rho M} \left(\sum_{i \in I_-} t_i + \sum_{i \in I_+} t_i \right) \\ &= \frac{1}{M} \sum_i \frac{t_i}{\rho} \end{aligned} \tag{5}$$

This final equation shows if we *adjust* all observed times for an algorithm / SUT combination by dividing by the success rate of that combination, then the arithmetic mean of these adjusted times will be the average *total* time taken, $\bar{t}_*$, which is our desired metric.

We perform this adjustment for both the number of visited states reported in Table 2, and for the elapsed time reported in Table 3 [1]. In both tables, lower values—fewer states visited or shorter elapsed times—are indicative of better time efficiency. If no adjustment is possible because none of the 30 trials were successful (i.e., $\rho = 0$), we set the number visited states and elapsed time to arbitrarily high values of 10^7 states and 1000 seconds, respectively.

In these tables, the row labelled 'Average (Geom. Mean)' is a measure of the average time efficiency of the algorithm across all 12 SUTs. Since the values vary by orders of magnitude between SUTs, we calculate this average by the following process using the geometric mean. For each i, where i is the index of a trial in a SUT / algorithm combination ranging from 1 to 30, the geometric mean is calculated of the i^{th} time responses across all 12 SUTs. The geometric mean of K values, $x_1, x_2, ..., x_K$, is:

$$\left(\prod_{j=1}^{K} x_j \right)^{\frac{1}{K}} \tag{6}$$

and thus, in contrast to the arithmetic mean, it considers multiplicative changes in the values: a change by any of the values by the same factor results the same changes in the geometric mean regardless of the magnitude of that value. Having calculated 30 geometric means in this way (one for each value of i) for the algorithm, the value reported for the algorithm in the row 'Average (Geom. Mean)' is the arithmetic mean of these 30 values.

[1] A dataset of the raw observations, including the unadjusted time data, is available from www.simonpoulding.net.

The sample of 30 geometric means for each algorithm is also the data summarised in the boxplots of Figures 3 and 4. One boxplot is shown for each of the algorithms, and summarises the distribution of the average performance of the algorithm across the 12 SUTs.

We discuss these results in section 4.7 below.

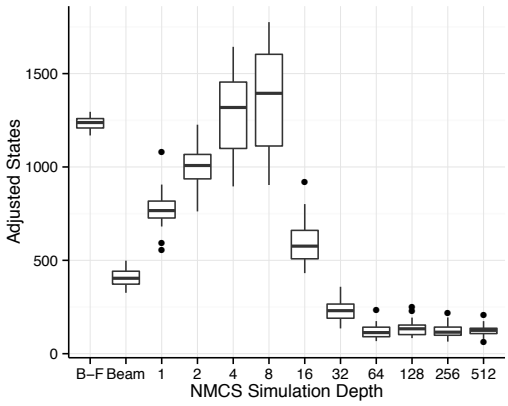

Figure 3: Boxplots of number of states visited (adjusted for success rate) for each algorithm, averaged across all SUTs

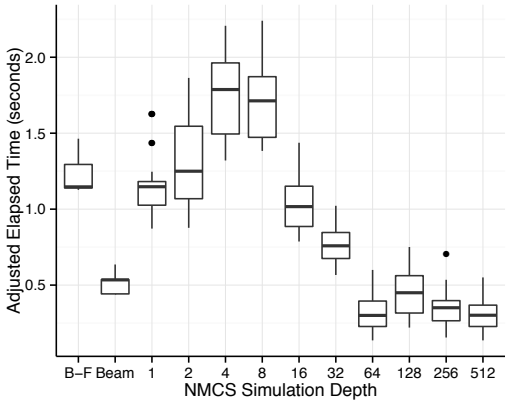

Figure 4: Boxplots of elapsed time in seconds (adjusted for success rate) for each algorithm, averaged across all SUTs

4.6.3 *Space Efficiency*

The space efficiency metric is the maximum memory consumed during a JPF trial. In table 4, the maximum value across the 30 trials in a SUT / algorithm combination is reported: lower values are better. The row 'Average (Geom. Mean)' summarises average memory usage of an algorithm across all 12 SUTs. It is calculated by forming a sample of 30 geometric means as for the time efficiency metrics, and then taking the maximum value. Figure 5 summarises the distribution of the average memory efficiency of each algorithm across the 12 SUTs.

SUT	B-F	Beam	N1,1	N1,2	N1,4	N1,8	N1,16	N1,32	N1,64	N1,128	N1,256	N1,512
boundedbuffer	1	1	1	1	1	1	1	1	1	1	1	1
clean	1	0.0667	0.1	0.1	0.367	0.5	0.467	0.7	0.7	0.6	0.633	0.533
deadlock	1	0.5	0.633	0.667	0.833	0.967	1	1	1	1	1	1
diningPhilosophers	1	1	1	1	1	1	1	1	1	1	1	1
groovy	1	0.333	0.533	0.467	0.567	0.5	0.9	1	1	1	1	1
log4j1	1	0.933	1	0.967	1	1	1	1	1	1	1	1
loseNotify	1	1	1	1	1	1	1	1	1	1	1	1
nestedmonitor	1	0.667	1	1	1	1	1	1	1	1	1	1
pool4	1	0.3	0.633	0.433	0.4	0.4	0.5	1	0.867	0.9	0.967	0.8
pool6	1	0.0333	0	0	0	0	0.0667	0.133	0.4	0.133	0.267	0.4
replicatedworkers	1	0	0.0667	0.133	0.0333	0.0667	0.1	0.2	1	1	1	1
sleepingBarber	1	1	1	1	1	1	1	1	1	1	1	1
Arithmetic Mean	1	0.569	0.664	0.647	0.683	0.703	0.753	0.836	0.914	0.886	0.906	0.894

Table 1: Mean success rate

SUT	B-F	Beam	N1,1	N1,2	N1,4	N1,8	N1,16	N1,32	N1,64	N1,128	N1,256	N1,512
boundedbuffer	235000	187	1240	1050	1600	3100	3540	3530	156	158	127	153
clean	79.8	4460	5580	8720	2850	3040	6360	4530	15600	25000	21600	31600
deadlock	262	62.9	94.8	125	140	123	35.8	40.7	49.3	38.7	37.7	34.8
diningPhilosoph'rs	969	95.5	177	229	356	466	407	100	51.1	61.8	59.5	62.7
groovy	243	236	155	263	336	581	282	45.6	50.6	43.5	39.8	50
log4j1	485	55.4	106	153	215	306	384	43.1	34.5	34.3	32.6	32.8
loseNotify	75.4	26.8	50.5	66.7	72.5	53.3	14.3	14.8	14.2	14.5	14	14.7
nestedmonitor	68.6	45.8	64.5	81.3	117	54.9	39.2	54.5	78.2	110	151	48
pool4	1020	390	182	436	769	1140	1020	106	588	410	160	750
pool6	13900	1470	1e+07	1e+07	1e+07	1e+07	11700	7870	1430	4220	2460	1410
replicatedwork'rs	545000	1e+07	56700	41900	278000	250000	308000	290000	3410	2760	2490	2970
sleepingBarber	590	72.5	140	201	289	409	346	29.5	27.6	27.8	28.1	28.1
Geometric Mean	1240	408	772	1010	1290	1350	601	231	119	136	119	124

Table 2: Mean number of states visited (adjusted for success rate)

SUT	B-F	Beam	N1,1	N1,2	N1,4	N1,8	N1,16	N1,32	N1,64	N1,128	N1,256	N1,512
boundedbuffer	23.7	0	0.667	0.833	0.967	1.13	1.43	1.83	0.0667	0.0333	0	0
clean	0	0	1.67	4.33	1.55	1	1.36	0.81	2.29	3.56	3.53	4.62
deadlock	0	0	0	0	0	0	0	0	0	0	0	0
diningPhilosophers	0.267	0	0	0	0	0.0333	0	0	0	0	0	0
groovy	0.0667	0	0	0.0714	0.176	0.0667	0	0	0	0	0	0
log4j1	1	0.536	0.833	1	1	1	1	0.433	0.433	0.867	0.233	0.0333
loseNotify	0	0	0	0	0	0	0	0	0	0	0	0
nestedmonitor	0.0333	0	0	0	0.0333	0	0	0	0.0667	0.1	0.167	0
pool4	1	0	0.0526	0.0769	0.583	1.08	1	0	0.231	0.148	0.0345	0.375
pool6	4	0	1000	1000	1000	1000	15	7.25	2	7	3.5	2.08
replicatedworkers	66.4	1000	43.5	25.5	129	78	69.3	54.3	6.2	5.27	4.93	5.5
sleepingBarber	1	0.333	0.133	0.233	1	1.43	0.967	1.23	0.1	0.1	0	0
Geometric Mean	1.2	0.521	1.17	1.29	1.76	1.72	1.05	0.768	0.322	0.45	0.341	0.307

Table 3: Mean elapsed time in seconds (adjusted for success rate)

SUT	B-F	Beam	N1,1	N1,2	N1,4	N1,8	N1,16	N1,32	N1,64	N1,128	N1,256	N1,512
boundedbuffer	603	123	370	156	156	225	226	156	156	123	123	123
clean	123	123	123	156	156	156	225	357	416	422	407	429
deadlock	123	123	123	123	123	123	123	123	123	123	123	123
diningPhilosophers	123	123	123	123	123	123	123	123	123	123	123	123
groovy	123	123	123	123	123	123	123	123	123	123	123	123
log4j1	156	123	123	123	123	123	123	123	123	123	123	123
loseNotify	123	123	123	123	123	123	123	123	123	123	123	123
nestedmonitor	123	123	123	123	123	123	123	123	123	442	304	123
pool4	156	123	123	123	123	123	123	123	156	156	123	156
pool6	370	123	123	123	123	123	123	123	123	123	123	123
replicatedworkers	967	123	457	481	466	476	533	544	889	910	864	910
sleepingBarber	123	123	123	123	123	123	123	123	123	123	123	123
Geometric Mean	189	123	150	140	143	147	152	153	163	161	169	161

Table 4: Maximum memory used in MB

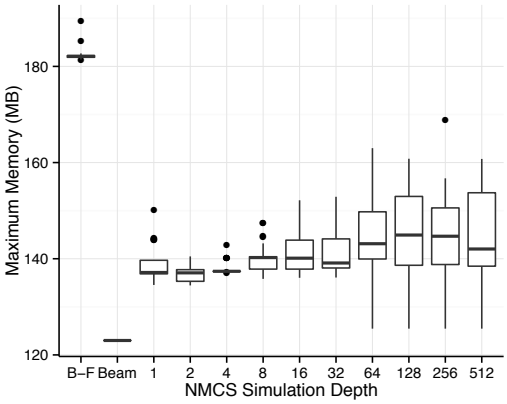

Figure 5: Boxplots of maximum memory used in MB for each algorithm, averaged across all SUTs

4.7 Discussion

4.7.1 RQ1: effect of simulation depth parameter

We first consider the NMCS algorithm time efficiencies, averaged across the SUTs. The number of visited states (Figure 3) shows a surprising pattern: as the NMCS simulation depth increases from 1 to 8, the number of states increases and therefore time efficiency become worse, but then rapidly improves as the simulation depth increases to 64. There is relatively little difference in the time efficiency using simulation depths of 64, 128, 256 and 512. A similar pattern for the elapsed time (Figure 4), showing that the pattern is not an artefact of our decision of how to calculate the number of visited states.

The implementation of beam search provided out-of-the-box by JPF might be thought of as NMCS with a simulation depth 0: the heuristic is calculated at the children of the state currently being explored with no simulation. We note, with interest, that with this interpretation of JPF's beam search, its time efficiency is consistent with the pattern seen for NMCS with simulation depth from 1 to 8.

Figure 3 and 4 demonstrate clearly that the simulation depth parameter has a very significant effect on the time efficiency. (For completeness: Kruskall-Wallis tests with the null hypothesis that the depth parameter has no effect on time efficiency gives p-values much less than 10^{-4} for both the number of visited states and the elapsed time.)

Table 4 shows that a similar pattern exists for each SUT individually. For some SUTs—such as `diningPhilosophers` and `loseNotify`—the deadlock can be found by visiting only a few tens of states, and therefore are often found during the first simulation by NMCS. The difference in time efficiency between, say, simulation depths of 8 and 64 is relatively small for these SUTs, but is still evident. However for SUTs for which the deadlock is much harder to find, the effect size is much greater. As an example, the deadlock in `replicatedworkers` is found by visiting approximately 73 times more states when the NMCS simulation depth is 8 than when it is 64.

Considering now the space efficiency in terms of the memory used by JPF, there is much less difference in the boxplots of Figure 5 for NMCS: the average memory used increases only slightly as the simulation depth increases, although at higher depth the spread of values of the distribution is larger. (A Kruskal-Wallis test here also gives a p-value much less than 10^{-4}.)

We conclude that the average time efficiency of NMCS depends on the simulation depth in a complex way: the best efficiency across these SUTs is achieved when the simulation depth is 64 or more. There is a much smaller effect of the simulation depth on space efficiency.

These experiments do not, by themselves, provide an explanation for the relationship between time efficiency and the NMCS simulation depth. The results in table 2 show that one contributing effect is that increasing the simulation depth increases the success rate, and so the total time required to find a deadlock—equation (2)—is reduced, but this is a trade-off since more states are visited when each simulation goes deeper.

For simpler SUTs such as `log4j1` and `losenotify`, it is apparent that the efficiency of NMCS at larger simulation depths is because deadlocks can be found easily once a certain depth is reached in the model. At these depths, the average number of states visited for these SUTs is less than the simulation depth and this suggests that a deadlock is typically found on the first random simulation to an appropriate depth. But this is not a general explanation. We performed an additional experiment using `replicatedworkers`—a more complex SUT where NMCS shows significant efficiency gains when the simulation depth increases to 64—that explored 10,000 random paths to depth 64. On none of these paths was a deadlock found, and so the efficiency of NMCS in this case is not because deadlocks are more easy to find at this depth.

4.7.2 RQ2: comparison to best-first & beam search

For RQ2, we compare the efficiency of NMCS with JPF's implementation of best-first search and beam search. To simplify the discussion, we make this comparison using the NMCS variant with a simulation depth of 64 as this gives close the best time efficiency across the 12 SUTs.

Figure 3 and Table 2 show that NMCS is significantly better than best-first search for most SUTs: the difference on average is a factor of approximately of 10.4 in terms of the number of visited states. (This difference is statistically significant: a Wilcoxon rank sum test gives a p-value of less than 10^{-4}.) We note that this difference is not true of all SUTs: for the SUT `clean`, best-first search is better than all variants of NMCS. Figure 4 and Table 3 show a qualitatively similar pattern in terms of elapsed time, although the pattern is more difficult to discern because JPF records elapsed time only to the nearest second and many trials finish in less than one second.

In general, JPF's beam search is much more time efficient than best-first search, but NMCS (again using simulation depth of 64 for this comparison) is still more time efficient by a factor of approximately 3.4 in terms of the number of visited states. (The difference is statistically significant according to a Wilcoxon rank sum test: the p-value is less than 10^{-4}.) A qualitatively similar pattern is seen in terms of elapsed time, but here NMCS is better by a factor of only approximately 1.6 (although again this figure may be unreliable owing to the reporting of elapsed time by JPF to only the nearest second).

Figure 5 and Table 4 shows that NMCS uses, on average, 25MB less memory than best-first search, but 40MB more memory than beam search, for the SUTs considered in these experiments. However, these differences have little practical impact given that JPF in this environment always used at least 123MB.

We conclude that, at the best settings of simulation depth, NMCS is more time efficient than both best-first search and beam search. It is less space efficient than beam search, but this difference has little practical difference.

5. RELATED WORK

We believe this work to be first application of Monte-Carlo Tree Search methods in model checking. However it is not the first application to software testing: Poulding and Feldt have recently used Nested Monte-Carlo Search to generate test data [9]. We note that in their application, it was not necessary to limit the depth of simulation: instead a limit was placed on the number of child states considered.

A number of researchers have applied metaheuristic search to the detection of concurrency bugs using model checking. For example, Godefroid and Khurshid [6], Alba et al. [1], and Shousha et al. [10] use evolutionary algorithms to construct a path that is then applied in the model in order to assess its fitness using the heuristic. In contrast, Staunton and Clark [11] use an Estimation of Distribution Algorithm to guide the selection of paths within the model itself; it is this work that is therefore closest to the approach taken in this paper.

6. CONCLUSION

In this paper we proposed the application of a Monte-Carlo Tree Search method, the Nested Monte-Carlo Search (NMCS) algorithm, as a heuristic search algorithm for model checking. We identified the need for an additional parameter to the standard NMCS algorithm to limit the length of the random simulations. We implemented the NMCS algorithm in Java PathFinder (JPF), and compared the time and space efficiency of the algorithm to JPF's best-first and beam search algorithms on 12 Java programs. The result showed that, on average, NMCS had superior time efficiency when the simulation depth was set to 64 or greater.

These results suggest a number of opportunities for further research.

Firstly, many of the SUTs used in the empirical work appear to be relatively simple: deadlocks could be found using most heuristic search algorithms in a few seconds. We wish to understand whether NMCS continues to demonstrate superior time efficiency for much larger models. The result provide some evidence that this might be case: the SUT `replicatedworkers` appears to be the most difficult for all of the algorithms, and for this SUT NMCS demonstrated a particularly large speed-up: approximately 160 times fewer states were visited by NMCS than by best-first search, and beam search never found a deadlock for this SUT.

Secondly, the best time efficiency for NMCS was achieved when the simulation depth parameter was 64 or greater. We intend to investigate whether the same limit is effective for a wider range of SUTs. One aspect of this investigation would be to investigate characteristics of the fitness landscape induced by the heuristic in the model, and whether the 'scale' of this landscape is related to the simulation depth.

Finally, we will investigate other model-checking heuristics. Although we have no reason to suspect that the NMCS algorithm is particularly suited to the 'most blocked' heuristics, we wish to replicate the results using other concurrency heuristics as well as structural metrics of type used by Groce and Visser in [8].

Acknowledgments

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

7. REFERENCES

[1] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding deadlocks in large concurrent Java programs using genetic algorithms. In *Proc. 10th Annual Conference on Genetic and Evolutionary Computation*, pages 1735–1742, 2008.

[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] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[6] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280. Springer, 2002.

[7] A. Groce and W. Visser. Heuristic model checking for java programs. In *Model Checking Software*, pages 242–245. Springer, 2002.

[8] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 12–21. ACM, 2002.

[9] 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)*, pages 1279–1286, 2014.

[10] M. Shousha, L. C. Briand, and Y. Labiche. A UML/MARTE model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. *IEEE Transactions on Software Engineering*, 38(2):354–374, 2012.

[11] J. Staunton and J. A. Clark. Searching for safety violations using estimation of distribution algorithms. In *Proc. 3rd International Workshop on Search-Based Software Testing*, pages 212–221, 2010.

[12] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.