Comparing Metaheuristic Algorithms for Error Detection in Java Programs

Francisco Chicano[1], Marco Ferreira[2], and Enrique Alba[1]

[1] University of Málaga, Spain
{chicano,eat}@lcc.uma.es
[2] Instituto Politécnico de Leiria, Portugal
mpmf@estg.ipleiria.pt

Abstract. Model checking is a fully automatic technique for checking concurrent software properties in which the states of a concurrent system are explored in an explicit or implicit way. The main drawback of this technique is the high memory consumption, which limits the size of the programs that can be checked. In the last years, some researchers have focused on the application of guided non-complete stochastic techniques to the search of the state space of such concurrent programs. In this paper, we compare five metaheuristic algorithms for this problem. The algorithms are Simulated Annealing, Ant Colony Optimization, Particle Swarm Optimization and two variants of Genetic Algorithm. To the best of our knowledge, it is the first time that Simulated Annealing has been applied to the problem. We use in the comparison a benchmark composed of 17 Java concurrent programs. We also compare the results of these algorithms with the ones of deterministic algorithms.

Keywords: Model checking, Java PathFinder, simulated annealing, particle swarm optimization, ant colony optimization, genetic algorithm.

1 Introduction

Software is becoming more and more complex. That complexity is growing for a variety of reasons, not the least of them is the need of concurrent and distributed systems. Recent programming languages and frameworks, such as Java and .NET, directly support concurrency mechanisms, making them an usual choice when developing concurrent and/or distributed systems. However, since these systems introduce interactions between a large number of components, they also introduce a larger number of points of failure. And this possible errors are not discoverable by the common testing mechanisms that are used in software testing. This creates a new need: to find software errors that may arise from the components communication, resource access and process interleaving. These are subtle errors that are very difficult to detect as they may depend on the order the environment chooses to execute the different threads, or components of the system. Some examples of this kind of errors are deadlocks, livelocks and starvation.

M.B. Cohen and M. Ó Cinnéide (Eds.): SSBSE 2011, LNCS 6956, pp. 82–96, 2011.
© Springer-Verlag Berlin Heidelberg 2011

One technique used to validate and verify programs against several properties like the ones mentioned is *model checking* [1]. Basically, a model checker uses a simplified implementation of the program, that is, a *model*, creating and traversing the graph of all the possible states of that model to find a path starting in the initial state that violates the given properties. If such a path is found it is a counterexample of the property that can be used to correct the program. Otherwise, if the algorithm used for the search of the counterexample is complete, the model is proven to be correct regarding the given properties.

The amount of states of a given concurrent system is very high even in the case of small systems, and it usually increases in a exponential way with the size of the model. This fact is known as *the state explosion problem* and limits the size of the model that a model checker can verify. Several techniques exist to alleviate this problem, such as partial order reduction [2], symmetry reduction [3], bitstate hashing [4] and symbolic model checking [5]. However, exhaustive search techniques are always handicapped in real concurrent programs because most of these programs are too complex even for the most advanced techniques.

When, even after state or memory reduction is somehow performed, the number of states becomes too big, two problems appear: the memory required to search for all states is too large and/or the time required to process those states is extremely long for practical purposes. That means that either the model checker will not be able to find an error nor prove the correctness of the model or, if it does find an error or prove the correctness of the model, it will not be in a practical run time. In those cases, the classical search algorithms like Depth First Search (DFS) or Breadth First Search (BFS), which are the most commonly used in model checking, are not suited.

However, using the old software engineering adage: "a test is only successful if it finds an error", we can think of model checking not as a way to prove correctness, but rather as a technique to locate errors and help in the testing phase of the software life cycle [6]. In this situation, we can stop thinking in complete search algorithms and start to think in not complete, but possibly guided search algorithms that lead to an error (if it exists) faster. That way, at least one of the objectives of model checking is accomplished. Therefore, techniques of bounded (low) complexity as those based on heuristics will be needed for medium/large size programs working in real world scenarios.

In this article we will study the behavior of several algorithms, including deterministic complete, deterministic non-complete, and stochastic non-complete search algorithms. In particular, the contributions of this work are:

- We analyze, compare and discuss the results of applying ten algorithms for searching errors in 17 Java programs.
- We include in the comparison algorithms from four different families of metaheuristics: evolutionary algorithms (two variants), particle swarm optimization, simulated annealing, and ant colony optimization.
- We use a simulated annealing algorithm (SA) for the first time in the domain of model checking.

– We use large Java models that actually pose a challenge for traditional model checking techniques and thus expand the spectrum of checkable programs.

The paper is organized as follows. In the next section we introduce some background information on heuristic model checking and Java PathFinder, which is the model checker used in this work. Section 3 presents a formal definition of the problem at hands. In Section 4 we briefly present the algorithms used in the experimental study and their parameters. Then, we describe the experiments performed and discuss the obtained results in Section 5. We conclude the paper in Section 6.

2 Heuristic Model Checking

The search for errors in a model can be transformed in the search for one objective node (a program state that violates a given condition) in a graph, the transition graph of the program, which contains all the possible states of the program. For example, if we want to check the absence of deadlocks in a Java program we have to search for states with no successors that are not end states.

Once we have transformed the search for errors in a search in a graph, we can use classical algorithms for graph exploration to find the errors. Some classical algorithms used in the literature with this aim are depth first search (DFS) or breadth first search (BFS). It is also possible to apply graph exploration algorithms that takes into account heuristic information, like A*, *Weighted* A*, *Iterative Deeping* A*, and *Best First Search*. When heuristic information is used in the search, we need a map from the states to the heuristic values. In the general case, this maps depends on the property to check and the heuristic value represents a preference to explore the corresponding state. The map is usually called *heuristic function*, that we denote here with h. The lower the value of h the higher the preference to explore the state, since it can be near an objective node.

The utilization of heuristic information to guide the search for errors in model checking is called *heuristic* (or *guided*) *model checking*. The heuristic functions are designed to guide the search first to the regions of the transition graph in which the probability of finding an error state is higher. This way, the time and memory required to search an error in a program is decreased on average. However, the utilization of heuristic information has no advantage when the program has no error. In this case, the whole transition graph must be explored.

A well-known class of non-exhaustive algorithms for solving complex problems is the class of metaheuristic algorithms [7]. They are search algorithms used in optimization problems that can find good quality solutions in a reasonable time. Metaheuristic algorithms have been previously applied to the search of errors in concurrent programs. In [8], Godefroid and Khurshid applied Genetic Algorithms in one of the first work on this topic. More recently, Alba and Chicano used Ant Colony Optimization [9] and Staunton and Clark applied Estimation of Distribution Algorithms [10].

2.1 Verification in Java PathFinder

There are different ways of specifying the model and the desired properties. Each model checker has its own way of doing it. For example, in SPIN [4] the model is specified in the Promela language and the properties are specified using Linear Temporal Logic (LTL). It is usual to provide the model checker with the properties specified using temporal logic formulas, either in LTL or CTL. It is also usual to find specific modelling languages for different model checkers. Promela, DVE, and SMV are just some examples. However, model checkers exist that deal with models written in popular programming languages, like C or Java. This is the case of Java PathFinder (JPF) [11], which is able to verify models implemented in JVM[1] bytecodes (the source code of the models is not required). The properties are also specified in a different way in JPF. Instead of using temporal logic formulas, the JPF user has to implement a class that tells the verifier algorithm if the property holds or not after querying the JVM internal state. Out of the box, JPF is able to check the absence of deadlocks and unhandled exceptions (this includes assertion violations). Both kind of properties belong to the class of *safety properties* [12].

In order to search for errors, JPF takes the `.class` files (containing the JVM bytecodes) and use its own Java virtual machine implementation (JPF-JVM in the following) to advance the program instruction by instruction. When two or more instructions can be executed, one of them is selected by the search algorithm and the other ones are saved for future exploration. The search algorithm can query the JVM internal state at any moment of the search as well as store a given state of the JVM and restore a previously stored state. From the point of view of the Java model being verified, the JPF-JVM is not different from any other JVM: the execution of the instructions have the same behaviour. The JPF-JVM is controlled by the search algorithm, which is an instance of a subclass of the `Search` class. In order to include a new search algorithm in JPF, the developer has to create a new class and implement the corresponding methods. This way, JPF can be easily extended; one aspect that is missing in other model checkers like SPIN. The role of the search algorithm is to control the order in which the states are explored according to the search strategy and to detect the presence of property violations in the explored states.

In JPF, it is possible to use search algorithms guided by heuristic information. To this aim, JPF provides some classes that ease the implementation of heuristic functions and heuristically-guided search algorithms.

3 Problem Formalization

In this paper we tackle the problem of searching for safety property violations in concurrent systems. As we previously mentioned, this problem can be translated into the search of a walk in a graph (the transition graph of the program) starting

[1] JVM stands for Java Virtual Machine.

in the initial state and ending in an objective node (error state). We formalize here the problem as follows.

Let $G = (S, T)$ be a directed graph where S is the set of nodes and $T \subseteq S \times S$ is the set of arcs. Let $q \in S$ be the *initial node* of the graph, $F \subseteq S$ a set of distinguished nodes that we call *objective nodes*. We denote with $T(s)$ the set of successors of node s. A finite walk over the graph is a sequence of nodes $\pi = \pi_1\pi_2 \ldots \pi_n$ where $\pi_i \in S$ for $i = 1, 2, \ldots, n$ and $\pi_i \in T(\pi_{i-1})$ for $i = 2, \ldots, n$. We denote with π_i the ith node of the sequence and we use $|\pi|$ to refer to the length of the walk, that is, the number of nodes of π. We say that a walk π is a *starting walk* if the first node of the walk is the initial node of the graph, that is, $\pi_1 = q$.

Given a directed graph G, the problem at hand consists in finding a starting walk π ($\pi_1 = q$) that ends in an objective node, that is, $\pi_* \in F$. The graph G used in the problem is the transition graph of the program. The set of nodes S in G is the set of states in of the program, the set of arcs T in G is the set of transitions between states in the program, the initial node q in G is the initial state of the program, the set of objective nodes F in G is the set of error states in the program. In the following, we will also use the words *state*, *transition* and *error state* to refer to the elements in S, T and F, respectively. The transition graph of the program is usually so large that it cannot be completely stored in the memory of a computer. Thus, the graph is build as the search progresses. When we compute the states that are successors in the transition graph of a given state s we say that we have *expanded* the state.

4 Algorithms

In this section we will present the details and configurations of the ten algorithms we use in the experimental section. In Table 1 we show the ten algorithms classified according two three criteria: completeness, determinism and guidance. We say that an algorithm is *complete* if the algorithm ensures the exploration of the whole transition graph when no error exists. For example, DFS and BFS are complete algorithms, but Beam Search and all the metaheuristic algorithms used here are non-complete algorithms. One algorithm is *deterministic* if the states are explored in the same order each time the algorithms is run. DFS and Beam Search are examples of deterministic algorithms, while Random Search and all the metaheuristics are non-deterministic algorithms. *Guidance* refers to the use of heuristic information. We say that an algorithm is *guided* when it uses heuristic information. A* and Beam Search are guided algorithms while Random Search and BFS are unguided algorithms.

For the evaluation of the tentative solutions (walks in the transition graph) we use the same objective function (also called *fitness* function) in all the algorithms. Our objective is to find deadlocks in the programs and we prefer short walks. As such, our fitness function f is defined as follows:

$$f(x) = deadlock + numblocked + \frac{1}{1 + pathlen} \tag{1}$$

Table 1. Algorithms used in the experimental section

Algorihm	Acronym	Complete?	Deterministic?	Guided?
Depth First Search [11]	DFS	yes	yes	no
Breadth First Search [11]	BFS	yes	yes	no
A* [11]	A*	yes	yes	yes
Genetic Algorithm [13]	GA	no	no	yes
Genetic Algorithm [13] with Memory Operator	GAMO	no	no	yes
Particle Swarm Optimization [14]	PSO	no	no	yes
Ant Colony Optimization [9]	ACOhg	no	no	yes
Simulated Annealing	SA	no	no	yes
Random Search	RS	no	no	no
Beam Search [11]	BS	no	yes	yes

where *numblocked* is the number of blocked threads generated by the walk while *pathlen* represents the number of transitions in the walk and *deadlock* is a constant which takes a high value if a deadlock was found and 0 otherwise. The high value that *deadlock* can take should be larger than the maximum number of threads in the program. This way we can ensure that any walk leading to a deadlock has better fitness than any walk without deadlock. All the metaheuristic algorithms try to maximize f.

The random search is a really simple algorithm that works by building limited-length random paths from the initial node of the graph. Then, it checks if an error was found in the path.

In the following we describe the SA algorithm, since it is the first time that this algorithm is applied to this problem (up to the best of our knowledge). We omit the details of the remaining algorithms due to space constraints. The interested reader should refer to the corresponding reference (shown in Table 1).

4.1 Simulated Annealing

Simulated annealing (SA) is a trajectory-based metaheuristic introduced by Kirkpatrick *et al.* in 1983 [15]. It is based on the statistical mechanics of annealing in solids. Just like in the physical annealing, SA allows the solution to vary significantly while the virtual temperature is high and stabilizes the changes as the temperature lows, freezing it when the temperature reaches 0. We show the pseudocode of SA in Algorithm 1.

SA works by generating an initial solution S, usually in some random form, and setting the temperature T to an initial (high) temperature. Then, while some stopping criteria is not met, SA randomly selects a neighbor solution N of S and compares its energy (or fitness) against the current solution's energy, getting the difference ΔE in temperature between them. The neighbor solution is accepted as the new solution if it is better than the current one or, in case it is worse, with a probability that is dependent on both ΔE and temperature T. SA then updates the temperature using some sort of decaying method. When the stopping criteria is met, the algorithm returns the current solution S.

Algorithm 1. Pseudo code of Simulated Annealing

```
1: S = generateInitialSolution();
2: T = initialTemperature;
3: while not stoppingCondition() do
4:    N = getRandomNeighbor(S);
5:    ΔE = energy(N) - energy(S);
6:    if ΔE > 0 OR random(0,1) < probabilityAcceptance(ΔE, T) then
7:       S = N
8:    end if
9:    T = updateTemperature(T);
10: end while
11: return S
```

The energy function in this case is the objective function f defined in Equation (1). Since we want to maximize this function (the energy), given an energy increase ΔE and a temperature T, the probability of acceptance is computed using the following expression:

$$probabilityAcceptance(\Delta E, T) = e^{\frac{\Delta E}{T}} \tag{2}$$

One critical function of the Simulated Annealing is the `updateTemperature` function. There are several different ways to implement this method. In our implementation we used a simple, yet commonly used technique: multiplying the temperature by a number α between 0 and 1 (exclusive). The smaller that number is, the faster the temperature will drop. However, if we detect a local maxima (if the solution isn't improved for a number of iterations) we reset the temperature to its initial value to explore new regions.

4.2 Parameter Settings

In a comparison of different kinds of algorithms one problem always poses: how to compare them in a fair way? This problem is aggravated by the fact that the algorithms work in fundamentally different ways: some algorithms search only one state at a time, some search for paths. Some check only one state per iteration, others check many more states per iteration, etc. This large diversification makes it very hard to select the parameters that make the comparison fair. The fairest comparison criterion seems to be the computational time available to each algorithm. However, this criterion would make it impossible to use the results in a future comparison because the execution environment can, and probably will, change. Furthermore, the implementation details also affect the execution time and we cannot guarantee that the implementations used in the experiments are the most effective ones. For this reason, we decided to established a common limit for the number of states each algorithm may expand. After a defined number of states have been expanded the search is stopped and the results can be compared.

In order to maintain the parameterization simple, we used the same maximum number of expanded states for every model even though the size of each model is

considerably different. We defined that maximum number of states to be 200 000, as it was empirically verified to be large enough to allow the algorithms to find errors even on the largest models. Having established a common value for the effort each algorithm may use, the parameterization of each individual algorithm can be substantially different from each other. For instance, we don't have to define the same number of individuals in the GA as the same number of particles in the PSO or as the same number of ants in the ACO. This gives us the freedom to choose the best set of parameters for each algorithm. However, in the case of the stochastic algorithms, and since this is a parameter that largely affects their execution, we have used the same heuristic function for all of them.

DFS, BFS and A* do not require any parameter to be set as they are generic, complete and deterministic search algorithms. For the metaheuristic algorithms, on the other hand, there are a variety of parameters to be set and although they could be optimized for each individual experiment, we have opted to use the same set of parameters for every experiment. These parameters were obtained after some preliminary experiments trying to get the best results for each particular algorithm. The parameters are summarized, together with the ones of RS and BS, in Table 2.

5 Experimental Section

In our experiments we want to verify the applicability of metaheuristic algorithms to model checking. We performed several experiments using the algorithms of the previous section and different Java implemented models. In order to determine the behavior of each search algorithm we have selected several types of models, including the classical Dining Philosophers toy model (both in a cyclic and a non-cyclic version), the more complex Stable Marriage Problem and two different communication protocols: GIOP and GARP. The Dining Philosopher models illustrate the common deadlock that can appear on multi-threaded algorithms. The difference of the cyclic and non-cyclic version is that while in the first one, called `phi`, each philosopher cycles through the pick forks, eat, drop forks and think states, in the non-cyclic version, called `din`, each philosopher only picks the forks, eats and drops the forks once, thus limiting the number of possible deadlocks. The Stable Marriage Problem (`mar`) has more interactions between threads and its implementation leads to a dynamic number of threads during executions. It contains both a deadlock and an assertion violation. Both the Dining Philosophers problem and the Stable Marriage Problem can be instantiated in any size (scalable), which makes them good choices to study the behavior of the search algorithms as the model grows. Finally, the communication protocols represent another typical class of distributed systems prone to errors. Both of these protocol implementations have known deadlocks which makes them suitable for non-complete search algorithms, because although they cannot prove correctness of a model, they can be used to prove the incorrectness and help the programmer to understand and fix the properties violations.

The results obtained from the experiments can be analyzed in several ways. We will discuss the results on the success of each algorithm in finding the errors,

Table 2. Parameters of the algorithms

Beam Search		**Random Search**	
Parameter	**Value**	**Parameter**	**Value**
Queue limit (k)	10	Path length	350

GeGA algorithm	
Parameter	**Value**
Minimum path size	10
Maximum path size	350
Population size	50
Selection operator	Tournament (5 individuals)
Crossover probability	0.7
Mutation probability	0.01
Elitism	true (5 individuals)
Respawn	after 5 generations with same population average fitness or 50 generations without improvement in best fitness

GeGAMO algorithm	
Parameter	**Value**
Minimum path size	10
Maximum path size	50
Population size	50
Selection operator	Tournament (3 individuals)
Crossover probability	0.7
Mutation probability	0.01
Elitism	true (3 individuals)
Memory operator frequency	10
Memory operator size	25
Respawn	after 5 generations with same population average fitness or 60 generations without improvement in best fitness

PSO algorithm		**ACOhg algorithm**	
Parameter	**Value**	**Parameter**	**Value**
Number of Particles	10	Length of ant paths	300
Minimum path size	10	Colony size	5
Maximum path size	350	Pheromone power (α)	1
Iterations Until Perturbation	5	Heuristic power (β)	2
Initial inertia	1.2	Evaporation rate (ρ)	0.2
Final inertia	0.6	Stored solutions (ι)	10
Inertia change factor	0.99	Stage length (σ_s)	3

SA algorithm	
Parameter	**Value**
Path size	350
Initial temperature	10
Temperature decay rate (α)	0.9
Iterations without improvement	50

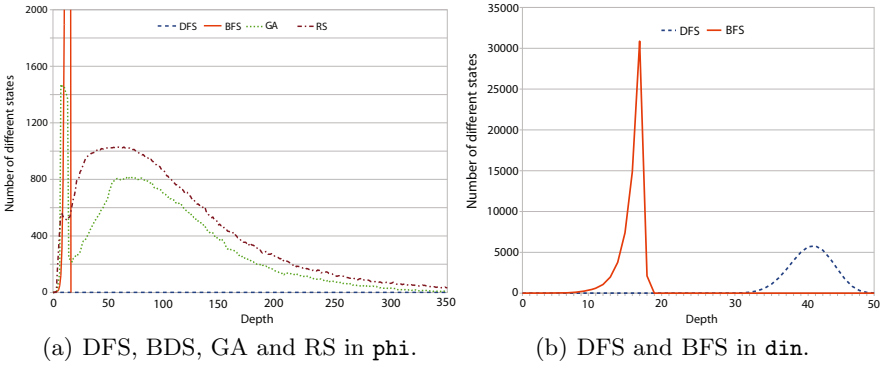
measured as the hit rate, and the length of the error trail leading to the error. Deterministic algorithms always explore the states in the same order, which means that only one execution per problem instance is needed. The results of stochastic search algorithms, however, could change at each execution. For this reason, each stochastic algorithm was executed 100 times per problem instance.

5.1 Hit Rate

We show the results of hit rate in Table 3. Regarding the Dining Philosophers cyclic problem (`phi`), we can observe that none of the exact search algorithms could find an error in the larger instances. In fact, all of them (DFS, BFS and A*) exhausted the available memory starting with 12 philosophers, while all of the stochastic search algorithms (GA, GAMO, PSO, SA, ACOhg and even RS) and BS had a high hit rate in all of the instances. To better understand the reason for

this, Figure 1(a) shows the distribution of the explored states after 200 000 states had been observed by two exact algorithms (DFS and BFS) and two stochastic algorithms (GA and RS). The remaining search algorithms were removed from the figure in order to have an uncluttered graphic. Figure 1(a) shows that DFS searched only one state per depth level (states explored in depths superior to 350 are not shown to maintain the graphic readability). That is coherent with the search algorithm which searches first in depth. However, this makes it difficult to find the error if it is not on the first transitions of each state. BFS, on the other hand, tried to fully explore each depth level before advancing to the next one. However, since the `phi` problem is a very wide problem (meaning that at each state there is a large number of possible outgoing transitions), the 200 000 states limit was reached quite fast. We can see a large difference in the behavior of the stochastic search algorithms. Both explore the search space both widely and in depth, simultaneously. This means that they avoid using all the resources in the few first depth levels, but also do not try to search too deep. Although they only visit the same number of states as their exact counterparts, they are spreader than the exact algorithms, which helps them find the error state. Considering that there are paths leading to errors in depths around 60, it is easy to see in Figure 1(a) why the stochastic algorithms found at least one of them while the exact algorithms missed them.

On the non-cyclic version of Dining Philosophers (`din`), DFS was able to detect errors in larger instances than the other exact algorithms. Figure 1(b) shows the reason why: since this problem is not cyclic, it ends after all the philosophers have had their dinner. Considering 12 philosophers, this happens, invariably, after 50 transitions. Since DFS has no more states to follow it starts to backtrack and check other transitions at the previous states. DFS concentrates its search at the end of the search space, and since the error state is at depth 36 (which is near the end), it is able to backtrack and explore other states at that depth level before consuming all the available memory. Figure 1(b) shows how



(a) DFS, BDS, GA and RS in `phi`. (b) DFS and BFS in `din`.

Fig. 1. Search behavior of algorithms. The X axis is the depth in the graph and the Y axis is the number of expanded states.

Table 3. Hit rate of the algorithms

Problem	DFS	BFS	A*	GA	GAMO	PSO	SA	ACOhg	RS	BS
phi 4	100	100	100	100	100	100	100	100	100	100
phi 12	0	0	0	100	100	100	100	100	100	100
phi 20	0	0	0	100	100	100	100	100	100	100
phi 28	0	0	0	100	100	100	100	100	100	100
phi 36	0	0	0	82	100	53	79	100	100	100
din 4	100	100	100	100	100	100	100	100	100	100
din 8	100	0	0	100	100	100	76	100	96	100
din 12	100	0	0	100	96	85	13	68	0	100
din 16	0	0	0	91	58	20	0	2	0	100
din 20	0	0	0	52	24	0	0	0	0	100
mar 2	100	100	100	100	100	100	100	100	100	100
mar 4	100	100	100	100	100	100	96	100	100	100
mar 6	100	0	0	100	100	100	100	100	100	100
mar 8	100	0	0	100	95	100	100	100	100	100
mar 10	100	0	0	100	25	100	100	100	100	100
giop	100	0	0	100	68	100	100	100	100	100
garp	0	0	0	100	2	80	87	87	100	0

many different states per depth level the DFS and BFS algorithms had checked after 200 000 expanded states. Although 200 000 were not enough for BFS to find the error, we can see that DFS was already exploring in the error state neighborhood. BFS, on the other hand, concentrates the search in the beginning of the search space and, after visiting 200 000 states, it is still far from the neighborhood of the error state. The `din` problem is much less forgiving than `phi`. There is only one error state, and one chance to find it. This creates an interesting problem: the probability that a random walk through the search space would find the error decreases substantially. In fact, our results show exactly that: RS starts to miss the error as the number of philosophers grow (and search space increases, therefore). All the metaheuristic search algorithms found the error in larger instances, but they too started to struggle to find it. Only the genetic algorithms found the error in all instances. Since these are the only algorithms in the set that mixes paths from different individuals to create new ones, it seems that they were able to find a pattern to reach the error, while the other stochastic algorithms have not. Finally, BS finds the error in all the cases.

Like the `din` problem, `mar` is also finite. This means that there is a (relatively small) limit on the maximum depth the search algorithm may look into. This knowledge, and the fact that DFS successfully found an error in all the problem instances may lead us to think that the problem is small and, therefore, simple. That is not, however, the case: the size of the search space grows exponentially with the size of couples. An interesting observation is that shape of the search space remains mainly unaffected with the growth of the number of couples, as seen in Figure 2. Considering the size of the search space, the good results of RS and the observations we made on the Dining Philosophers problems, it seems that the `mar` problem have many different paths leading to an error state. To

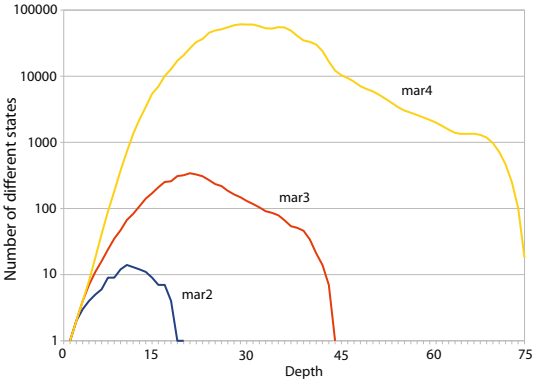


Fig. 2. Search space for the `mar` problem as seen by DFS for instances of 2, 3 and 4 couples. Note that the Y axis is logarithmic.

check this hypothesis we have checked all of the search space for the `mar` problem with 3 couples and found that with only 5156 different states in the search space, there are 30 error states and 216 different paths leading to those error states. This is indeed a large number of paths for the search space size and supports our hypothesis.

In general, when searching for errors, we have observed that non-complete algorithms have a higher success rate than the tested complete algorithms. From figures 1(b) and 2 we can observe that one reason seems to be the concentration of the search in either the beginning of it (BFS) or the end of it (DFS). Pure stochastic algorithms, like random search, explores a wider portion of the space at all the allowed depths (our random implementation is depth bounded). The metaheuristic algorithms tend to explore more through the search space and exploit areas that seem interesting, which typically are neither at the beginning or at the end of the search space. Complete search algorithms start to fail to search the whole search space very soon. Are they really applicable for software model checking? If some form of abstraction can be used, then maybe. However they still require large amount of memory and they are not checking the final implementation, so specific implementation details could still violate the properties checked before. However, failing to prove the model correctness does not mean they cannot be used to find errors. The `mar` problem is a good example of that. The search space could not be completely verified after using only 4 couples, but DFS was able to find errors even with 10 couples. The ability of DFS and BFS to find errors in large search spaces depends not so much on the size of the search space but more on its shape, as shown in the `din` and `mar` tests.

5.2 Length of the Error Trails

The length of the error trail for each algorithm is also an important result that must be compared. Since the error trail is the information the developers have to debug the application, the more concise it is, the better. So, shorter error

trails means that less irrelevant states exists in the trail, making it easier for the developer to focus on what leads to the errors As with the hit rate, exact algorithms always return the same error trail for each problem. Stochastic algorithms do not, so we present in Table 4 both the averages of the length of the first error trail found and of the shortest error trail length found in the 100 executions. We include both of these values because the stochastic algorithms do not stop after an error has been found, but only after 200 000 states have been expanded, which means that more than one path to an error may be found during the search.

Table 4. Length of the error trails (first/shortest)

Prob.	DFS	BFS	A*	GA	GAMO	PSO	SA	ACOhg	RS	BS
`phi 4`	169	16	16	48/16	28/16	52/16	47/16	71/16	50/16	22
`phi 12`	–	–	–	149/52	78/58	173/55	178/70	175/72	193/62	74
`phi 20`	–	–	–	220/116	131/114	248/126	251/140	244/163	319/135	224
`phi 28`	–	–	–	267/192	188/174	275/210	278/216	351/268	504/227	393
`phi 36`	–	–	–	283/269	248/232	282/278	290/274	495/381	616/324	753
`din 4`	12	12	12	12/12	12/12	12/12	12/12	12/12	12/12	12
`din 8`	24	–	–	24/24	24/24	24/24	24/24	24/24	24/24	24
`din 12`	36	–	–	36/36	36/36	36/36	36/36	36/36	–	36
`din 16`	–	–	–	48/48	48/48	48/48	–	48/48	–	48
`din 20`	–	–	–	60/60	60/60	–	–	–	–	60
`mar 2`	14	12	12	13/12	13/12		13/12	12/12	13/12	12
`mar 4`	63	26	28	42/27	39/27	43/27	45/29	42/28	44/29	36
`mar 6`	140	–	–	91/48	75/56	94/51	93/57	93/56	92/55	59
`mar 8`	245	–	–	148/77	108/94	154/79	149/91	152/92	151/88	172
`mar 10`	378	–	–	199/109	154/142	215/114	207/127	214/130	223/125	260
`giop`	232	–	–	251/238	252/247	263/236	264/243	256/242	284/239	355
`garp`	–	–	–	184/115	123/121	184/128	204/147	305/278	245/111	–

In all the problems, BFS shows the shortest possible error trail. However, as we have seen in the previous section, it starts to fail to find the error very quickly, so it cannot be used as a base comparison value for all the problem instances. DFS, on the other hand, finds the error in larger instances but the error trails provided by this algorithm are the largest of all the algorithms, with one exception: the `giop` problem. In this problem, the shortest error trail consists on always choosing the left-most transition available in each state, which is exactly the behavior DFS always present. The A* behaves mostly like BFS, again with one strange exception: in the `mar4` problem, the error trail is not the smallest one possible. This means that the heuristic we used is not very good for this problem, as it slightly misleads A*. This is also the reason for the low hit rate of A*. BS is very effective at finding the error but the length of the tail is usually far from the optimum, except for some small instances.

Among the stochastic algorithms, the genetic algorithms seem to be the ones that provide the shortest error trails. There is a difference in behavior between the GA using the memory operator (GAMO) and not using it. While the GA

usually finds the shortest error trail, the length of the error trail for the first error found is usually smaller using the memory operator, and is not too far from the shortest error trail. This means that if we change the stopping criteria of the algorithms in order to stop as soon as an error is found, then using the memory operator seems to be a better choice. All the guided stochastic algorithms show good results both on the first error trail and the shortest error trail. RS, which is not guided, also shows good results in these problems when we consider the shortest error trails only. However, the length of the first error trail found by RS is usually much larger than the ones found by the other algorithms.

6 Conclusion and Future Work

In this work we presented a comparison of five metaheuristic algorithms and five other classical search algorithms to solve the problem of finding property violations in concurrent Java programs using a model checking approach. We used a benchmark of 17 Java programs composed of three scalable programs with different sizes and two non-scalable programs. We analyzed the results of the algorithms in terms of efficacy: the ability of a search algorithm to find the property violation and the length of the error trail. The experiments suggests that metaheuristic algorithms are more effective to find safety property violations than classical deterministic and complete search algorithms that are commonly used in the explicit-state model checking domain. They also suggest that non-complete guided search algorithms, such as Beam Search, have some advantages against both guided and non-guided complete search algorithms such as A* and DFS. Finally, these experiments also suggest that distributing the search effort in different depths of the search space tends to raise the efficacy of the search algorithm.

As future work we can explore the possibility of designing hybrid algorithms that can more efficiently explore the search space by combining the best ideas of the state-of-the-art algorithms. We can also design stochastic complete algorithms that are able to find short error trails in case error exist in the software and can also verify the program in case no error exists.

Acknowledgements. This research has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project) and the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
2. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. International Journal on Software Tools for Technology Transfer (STTT) 2(3), 279–287 (1999)

3. Lafuente, A.L.: Symmetry reduction and heuristic search for error detection in model checking. In: Workshop on Model Checking and Artificial Intelligence (2003)
4. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2004)
5. Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (1994)
6. Bradbury, J.S., Cordy, J.R., Dingel, J.: Comparative assessment of testing and model checking using program mutation. In: Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION 2007), Windsor, UK, pp. 210–222 (2007)
7. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Comput. Surv. 35(3), 268–308 (2003)
8. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. International Journal on Software Tools for Technology Transfer 6(2), 117–127 (2004)
9. Alba, E., Chicano, F.: Finding safety errors with ACO. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1066–1073. ACM Press, London (2007)
10. Staunton, J., Clark, J.A.: Searching for safety violations using estimation of distribution algorithms. In: International Workshop on Search-Based Software Testing, pp. 212–221. IEEE Computer Society, Los Alamitos (2010)
11. Groce, A., Visser, W.: Heuristics for model checking java programs. International Journal on Software Tools for Technology Transfer (STTT) 6(4), 260–276 (2004)
12. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc., New York (1992)
13. Alba, E., Chicano, F., Ferreira, M., Gomez-Pulido, J.: Finding deadlocks in large concurrent java programs using genetic algorithms. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1735–1742. ACM, New York (2008)
14. Ferreira, M., Chicano, F., Alba, E., Gómez-Pulido, J.A.: Detecting protocol errors using particle swarm optimization with java pathfinder. In: Smari, W.W. (ed.) ISHPC 2000, pp. 319–325 (2008)
15. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)