Generating Counter-examples through Randomized Guided Search

Neha Rungta and Eric G Mercer

Department of Computer Science
Brigham Young University
Provo, UT 84602, USA

Abstract. Computational resources are increasing rapidly with the explosion of multi-core processors readily available from major vendors. Model checking needs to harness these resources to help make it more effective in practical verification. Directed model checking uses heuristics in a guided search to rank states in order of interest. Randomizing guided search makes it possible to harness computation nodes by running independent searches in parallel in a effort to discover counter-examples to correctness. Initial attempts at adding randomization to guided search have achieved very limited success. In this work, we present a new low-cost randomized guided search technique that shuffles states in the priority queue with equivalent heuristic ties. We show in an empirical study that randomized guided search, overall, decreases the number of states generated before error discovery when compared to a guided search using the same heuristic. To further evaluate the performance gains of randomized guided search using a particular heuristic, we compare it with randomized depth-first search. Randomized depth-first search shuffles transitions and generally improves error discovery over the default transition order implemented by the model checker. In the context of evaluating randomized guided search, a randomized depth-first search provides a lower bound for establishing performance gains in directed model checking. In the empirical study, we show that with the correct heuristic, randomized guided search outperforms randomized depth-first search both in effectively finding counter-examples and generating shorter counter-examples.

1 Introduction

The current trend in micro-processor design is to group multiple processors into a single silicon die and package. For example, dual-core processors are quickly becoming mainstream, and quad-core packages are readily available from most vendors. CEO Paul Otellini, at a recent Intel development forum, displayed an 80 core prototype chip capable of terabyte per second data exchange and pledged production runs in the next five years [25]. The trend is clearly to put more processors on a single die rather than to increase clock speed and computation in a single processor. This is leading to an explosion in computational resources.

The question for the model checking community given the growth in multi-core processors, as well as parallel and distributed systems, is how can we harness this computation power? At the heart of explicit state model checking is an exhaustive proof to show the absence of a specific behavior. The proof literally enumerates, in a largely brute-force manner, the entire behavior space of the system being verified [4]. The complexity of the systems, however, limits practical application of model checking in both time and space. Aggregating the available computation resources to solve the model checking problem can help to improve the situation.

Parallel and distributed model checking has shown some limited promise in utilizing large amounts of computation resources [35, 21, 1, 20, 3, 19]. The focus of the community is to find ways to harness several computation nodes to cooperatively construct the exhaustive proof. These approaches generally look appealing in low node counts but are less efficient as more computation nodes are added [22]. Seminal work goes so far as to prove that depth-first search itself is inherently sequential and does not lend itself to parallel computation [29]. This may explain the lack of scaling in current approaches and possibly suggest that we need a fundamentally different algorithm for model checking that is less sequential and more amenable to parallelization.

As a counterpoint, it is possible to parallelize model checking by moving away from an exhaustive proof and instead focus on counter-example generation. In other words, run several independent experiments with some degree of randomization on individual computation nodes to find a counter-example to the proof. This is in contrast to several computation nodes cooperatively constructing an exhaustive proof. The shift in focus from exhaustive proof to counter-example generation began in the directed model checking community, and it opens new avenues for distributed model checking.

Early researchers of parallel and distributed model checking explored the concept of random walk for counter-example generation with modest success [17, 34, 24]. Random walk has inherently low memory requirements, and the work distributes these random walk based searches over many computation nodes in hopes of discovering a counter-example. The effectiveness of random walk in terms of coverage is critically dependent on the structure of the model [28, 2, 18]. Empirical studies show that random walk is not very useful for error discovery in the models where it achieves poor coverage. This creates a need for effective randomized searches which better harness the computation resources.

Recent work studying default search order in model checker performance contributes a key insight to randomization of a regular depth-first search [7]. Controlling for default search order in depth-first search by randomly choosing transitions to explore (randomized DFS) dramatically improves counter-example generation [6]. Independent randomized DFS searches easily distribute to any number of computation nodes, however, like any search method, randomized DFS breaks down in certain models [32]. The issue in randomized DFS is that it blindly moves through the behavior space even when there is informa-

tion readily available about the structure of the model and the property being invalidated that can improve the search.

Directed model checking uses heuristics to rank interest in states and guide the search of the behavior space to efficiently generate counter-examples [37, 9, 10, 16, 27, 33, 8, 31]. The heuristics generally consider either the model structure or the property being validated to rank the states. A guided search then orders the states in a priority queue based on the path cost and heuristic ranking where states estimated to lead more quickly to a counter-example are explored before other states. Guided search is effective in counter-example generation and often succeeds where depth-first search fails. More importantly, the length of the counter-examples generated by guided search algorithms are often shorter than those generated by depth-first search. This simplifies the developer's task of understanding the counter-example.

Guided search also benefits from randomization, and like depth-first search, once randomized, it can be run independently in parallel (randomized GDS[1]). Preliminary work in randomized GDS chooses randomly from the first n-best entries of the priority queue when selecting the next state to explore [23]. The effectiveness of the randomization is not clear from the empirical study. In some instances, the randomization helps; while in other instances, the randomization hurts. The control, n, in [23] only ranges over a limited set of values between two and five, and the algorithm also does not distinguish between states in the priority queue with different heuristic values. In Java PathFinder v4.0 (JPF), it is also possible to execute a randomized GDS by randomizing the transition order in generating successors before adding them to the priority queue. This randomization, however, has very limited impact on the actual default search order in the guided search. Clearly, there are several open questions in randomized GDS left to be explored.

This paper presents a new randomized GDS algorithm that completely shuffles states in the priority queue with equal heuristic rankings. We show that full randomization of the guided search improves the effectiveness of the search over default search order in an empirical study. The empirical study uses characterized benchmarks from [7, 32] and published heuristics for the JPF, [36], and Estes, [26], model checkers. This paper also presents a second empirical study on the new randomized GDS algorithm in context of randomized DFS using the previously mentioned models and heuristics. The second study highlights the role of the heuristic in performance. When the heuristic is correctly matched to the models and properties, the new randomized GDS algorithm outperforms randomized DFS in both the effectiveness of the search in finding counter-examples and the length of the counter-examples. When the heuristic is not correctly matched to the models or properties, randomized DFS is more effective in error discovery which demonstrates a need to develop better heuristics for those classes of models and properties.

[1] We use randomized GDS to refer generally to any algorithm that adds randomization into guided search, and we will clearly indicate how the search is randomized in the context in which it appears.

The algorithm and empirical studies in this paper underscore a need to develop methods that match heuristics to models and the properties being disproved. This work and other work such as [23] and [6] also revisit a new way to view randomization, model checking, and search techniques. It motivates a need to study and understand how to best use randomization in model checking and parallelization for counter-example generation. Research in this area is especially timely given the rapid increase in computational resources, and more importantly, the ever increasing need for practical model checking in system design.

2 Background

It is important to control for default search order when evaluating model checking algorithms because implementation details in the model checker itself affect performance to a larger degree than previously supposed [7]. For example, in a simple depth-first search, the state at the top of a search stack may have several enabled transitions that move the current state to the next state of computation. The choices arise from non-determinism in the model, where the non-determinism is usually a result of scheduling decisions or input locations. The principle observation in [7] is that controlling for the default order in which a model checker selects transitions during depth-first search dramatically affects the outcome of counter-example generation. The work in [7] proposes a randomized DFS that controls for default transition order by shuffling transitions enabled at each state. Follow-on work in [6] shows that randomized DFS is effective in counter-example generation across their benchmark set[2]. In the words of [7], "*[T]hese findings tell a strong cautionary tale*", because default search order significantly affects performance of the techniques being evaluated in comparison studies. This is especially critical for directed model checking which relies on comparison studies to establish performance gains.

Directed model checking uses a guided search rather than depth-first or breadth-first search to find counter-examples for the property being verified. The fundamental assumption is that an error does exist in the model, and the goal is to find the error before exhausting computation resources. The work in this paper focuses on a greedy best-first search; although, the ideas are equally applicable to other best-first search techniques that make no guarantee on the optimality of the counter-example. In other words, the results of an A^* search are not significantly affected by our approach. A greedy best-first search is illustrated in Fig. 1. The top state in Fig. 1 is the initial state. At each iteration of the search, a state is removed from a priority queue, its successors are generated, ranked by a heuristic function, and inserted into the priority queue. For example, the initial state in Fig. 1 has three successors which are ranked 12, 9, and 2. These states are inserted into the priority queue. The next iteration of the search

[2] There are other default orders in model checkers that are yet to be controlled as evidenced in [32], where different versions of JPF yield different results in randomized DFS.

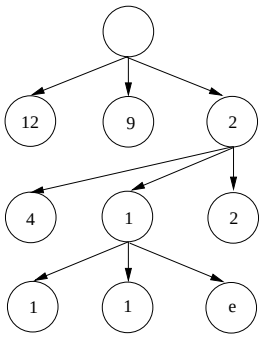

Fig. 1. An illustration of greedy best-first search that chooses the state nearest to the goal state to expand in the search based on a heuristic function.

removes the state with rank 2 from the priority queue and repeats the process. The heuristic function estimates the nearness of a state to an actual goal state. The goal state in our example is marked with the 'e' character. The goal state in directed model checking is an error state from which we build a counter-example to the specified property. A good heuristic for a greedy best-first search often converges quickly to an error state, and the length of the counter-example is near minimal.

Directed model checking critically relies on empirical studies to show performance gains over depth-first search, and like depth-first search, must control for default search order. For example, consider a priority search queue that contains over 100,000 states and a heuristic function that assigns an integer value between one and six to each state. Invariably, there are many thousand states with equivalent heuristic values. The order in which they are explored is largely controlled by the order in which they are generated by the model checker and ordered in the priority queue. During a guided search, some function compares the heuristic value of a newly generated state to the heuristic values of existing states in the queue before inserting the new state in the queue based on its ranking. Most often, this function uses a pre-determined ordering to sort states that have the same heuristic value. For example, when comparing a newly generated state, s_1, with a heuristic value, x, to an existing state in the priority queue, s_2, with a heuristic value, x, the state ordering function always inserts state s_1 after s_2 in the priority queue. The order in which states s_1 and s_2 are explored can potentially affect the total number of states generated before error discovery—a fact disregarded by the ordering function. The lesson from [7] is that these default choices in the model checker need to be controlled. This gives rise to randomized GDS which in the context of this paper refers to a greedy best-first search with some randomization to control for default order.

There are several ways to implement randomized GDS, and each controls for default order in the priority queue to a certain extent. For example, [23] randomly chooses between the n-best entries in the priority queue, and JPF v4.0 allows

the transition order to be shuffled during state generation. The former method shows some potential while the later method is not effective in randomization. This paper presents a new algorithm for randomized GDS that controls for all heuristic ties in the priority queue. We show that with the correct heuristic function, our new algorithm for randomized GDS outperforms not only the greedy best-first search using default ordering but randomized DFS as well. This is especially true in models that are hard—that is, models where randomized DFS is not successful.

3 Randomized GDS

Current techniques for randomization of guided search are not effective in exploiting the full potential of the randomization. For example, as mentioned previously, the approach presented in [23] limits the randomization to the n-best entries in the priority queue, where n is specified by the user. As another example, JPF allows for randomization in its searches. To understand its approach, we need to first look at its priority queue implementation; specifically, the `DefaultComparator` class. The class uses state identifiers and hash values to resolve heuristic ties between states in the priority queue. The state identifiers and hash values map to the same states in every single run of a guided search and deterministically resolve the heuristic ties. Turning on the `randomize_choices` option in JPF successfully modifies the order in which successors, for a particular state, are added to the priority queue because the successors are now assigned different state identifiers every time we execute a guided search trial. This randomized GDS approach causes only a small amount of variance in the number of states generated before error discovery when compared to the guided search since the randomization is limited to the successors of a given state. Our studies show that the limited amount of randomization is not effective in significantly changing the default search order.

To fully exploit the potential of randomization in directed model checking we define a randomized GDS algorithm that randomly shuffles states with equivalent heuristic ranking in the priority queue. The pseudo-code for this algorithm is presented in Fig. 2. The algorithm is *embarrassingly parallel* [15]. Several trials of the new randomized GDS algorithm can be launched in parallel on different computation nodes since each randomized GDS trial is completely independent of the other trials. There is no communication overhead between the trials which allows the algorithm to scale up to an arbitrary number of computation nodes.

In the randomized GDS algorithm, we associate a random value with each state generated during model checking in addition to its heuristic value. The tuple $\langle s_i, h_i, r_i \rangle$ in Fig. 2 is an element stored in the priority queue where s_i is the state, h_i is the heuristic ranking of s_i, and r_i is the random value associated with s_i. The randomized GDS algorithm employs a new comparator function, compare_vals, that is also shown in Fig. 2 and uses the random values as a secondary key to sort states with the same heuristic rankings. The approach enables us to effectively randomize the order of states with same heuristic values

```
   /* N is the set of computation nodes */
procedure randomized_guided_search_init(N)
   for each i ∈ N do
      execute(randomized_guided_search(), i)
   wait_for_all_nodes_to_terminate_execution()
   gather_results(1...N)
   return

   /* Add initial element ⟨s0, h0, r0⟩ to PriorityQueue PQ */
   /* Add s0 to the Visited set */
procedure randomized_guided_search()
   while PQ ≠ ∅ do
      ⟨si, hi, ri⟩ := PQ.dequeue()
      for each s' ∈ successors(si) do
         if error(s') then
            return Error Statistics
         if s' ∉ Visited then
            Visited := Visited ∪ {s'}
            PQ.enqueue(⟨s', heuristic(s'), rand_val()⟩)
   return No Errors Found

   /* PriorityQueue PQ uses compare_vals to order states */
procedure compare_vals(⟨s1, h1, r1⟩, ⟨s2, h2, r2⟩)
   if h1 > h2 then
      return true
   else if h1 < h2 then
      return false
   else
      if r1 > r2 then
         return true
      else
         return false
```

Fig. 2. Pseudo-code for randomized GDS that shuffles states with the same heuristic values using a secondary key from a random number generator.

across different states and search levels. The new randomized GDS algorithm has a low cost of randomization because maintaining the random value is the only additional cost it incurs when compared to a regular guided search.

We present two empirical studies that compare randomized GDS to default order guided search. The first study is in JPF v4.0 uses Java benchmarks and the second study is in Estes uses C benchmarks. JPF contains a suite of structural heuristics, [16], that exploit thread properties in Java programs and also has a heuristic for finding feasible abstract counter-examples [27, 16]. The Java models used in this study are small to medium sized programs that contain concurrency errors. These models have been collected from different sources: original papers presenting the heuristics [16], concurrency literature [12], research describing Java specific errors [14], and the IBM benchmark suite [13]. Additionally, these

models are characterized to a certain degree having been used recently in two extensive benchmarking studies [7, 32].

Our empirical study is conducted on a super-computing cluster with 618 nodes. We conduct a single experiment of executing 100 trials of our randomized GDS algorithm in parallel for each subject in the study. The choice of 100 trials is arbitrary, but we believe its size is sufficient to indicate general trends in performance. The randomized GDS trials and the guided search are allocated 7GB RAM, and the execution time is bounded at 1 hour. The 1 hour is again arbitrary but together with 100 trials constitutes an upper bound of 100 hours of computation for each model—a significant amount of resources.

Table 1 is a comparison between the default order guided search and our new randomized GDS algorithm in JPF. We present results for four different heuristics in JPF: choose-free heuristic, most-blocked heuristic, interleaving heuristic, and the prefer-thread heuristic. Based on the description of the heuristics in [16] and our knowledge of the models, we pick heuristics that are most likely to work well for a given model. We present, in Table 1, the number of states generated for a default order guided search (`GDS`). The values in Table 1 with the form, x^*, indicate that the search generated x number of states before running out of either time or memory. For the new randomized GDS algorithm (`Randomized-GDS`), in Table 1, we present the following statistics: path error density (`PED`), minimum (`Minimum`) and maximum (`Maximum`) number of states generated in a single error discovering randomized GDS trial among all the trials, mean (`Mean`) number of states generated in all the error discovering randomized GDS trials, and the 95% confidence interval (`95% CI`) for the mean number of states. The path error density is the ratio of the number of error discovering randomized GDS trials to the total number of trials executed.

The results in Table 1 show that the new randomized GDS algorithm, overall, improves the error discovery for a given heuristic over default search order. In the `AccountSubtype(2,2)` model, the default order guided search does not find an error even after exploring over 2.22 million states. In contrast, all 100 trials of the new randomized GDS algorithm find an error and explore only 193, 313 states—on average—before error discovery. Furthermore, the maximum number of states generated—642,193—by a single randomized GDS run of the new algorithm is also dramatically lower than the number of states generated by the default order guided search. Similar behavior is observed in all the `ProducerConsumer` models, and some `TwoStage`, `Piper`, and `Wronglock` models. In certain models, the mean number of states generated by the new randomized GDS algorithm is more than the states generated by the default order guided search, as seen in the `Deos(abstracted)` and `Reorder(1,5)` models; however, even in these models, the minimum number of states generated by the new randomized GDS algorithm is less than the number of states generated by the default order guided search.

Table 2 presents the results of running our new randomized GDS algorithm on different distance heuristic functions implemented in the Estes model checker [26]. We evaluate three specific distance heuristic functions in Table 2: FSM [11], EFSM [30], and e-FCA [31]. The only change in the setup for evaluating heuris-

Table 1. Comparing the performance of default order guided search (GDS) and randomized guided search (Randomized-GDS) using the heuristics in JPF and published benchmarks.

Model	GDS	Randomized-GDS				
		PED	Minimum	Mean	Maximum	95% CI
ChooseFree Heuristic						
Deos(abstracted)	16	1.00	11	40	423	14
RwNoExcpChk(2,100,1)	372,826	1.00	769	6,419	20,865	739
MostBlocked Heuristic						
Clean(1,1,12)	188	1.00	33	377	993	59
Piper(2,2,2)	16,437	1.00	240	1,338	3,909	171
Piper(2,4,4)	$2,478,360^*$	0.87	138,916	1,229,530	2,274,249	116,015
Interleaving Heuristic						
Raxextended(4,3)	$1,225,743^*$	1.00	404	20,774	670,813	14,480
PreferThreads Heuristic						
Accountsubtype(2,2)	$2,225,914^*$	1.00	30,726	193,313	642,193	94
Producerconsumer(1,10,4)	$1,783,620^*$	0.93	2,774	145,466	742,693	36,519
Producerconsumer(1,12,4)	$1,781,899^*$	0.90	13,830	238,092	960,610	52,981
Producerconsumer(1,16,4)	$1,781,530^*$	0.49	7,280	257,131	889,248	67,850
Producerconsumer(1,8,4)	$1,835,216^*$	1.00	1,148	156,428	925,537	38,689
Producerconsumer(2,2,4)	$2,591,457^*$	1.00	10,902	109,394	313,929	13,602
Producerconsumer(2,4,4)	$2,016,936^*$	1.00	2,592	213,491	1,122,008	45,523
Producerconsumer(2,8,4)	$1,721,824^*$	0.68	21,055	434,401	1,098,461	77,976
Reorder(1,1)	144	1.00	40	98	163	6
Reorder(1,5)	545	1.00	36	14,864	64,447	4,312
Reorder(10,1)	1,727,521	0.00	-	-	-	-
Reorder(5,1)	15,207	1.00	393	10,850	30,790	1,473
Reorder(8,1)	274,125	0.80	10,789	714,454	2,624,613	120,013
Reorder(9,1)	691,264	0.32	324,035	861,445	1,412,937	110,618
Twostage(1,1)	218	1.00	53	134	246	9
Twostage(2,5)	24,187	0.96	218	361,571	1,681,177	97,480
Twostage(5,2)	322,593	0.96	5,419	417,841	2,170,752	95,440
Twostage(6,1)	716,413	0.94	31,346	486,830	1,626,718	76,994
Twostage(7,1)	$2,354,460^*$	0.36	81,218	867,382	1,411,624	120,191
Twostage(8,1)	$2,119,657^*$	0.05	178,476	755,151	1,259,085	514,492
Wronglock(1,1)	156	1.00	37	67	122	4
Wronglock(1,10)	7,391	1.00	94	98,616	1,805,704	58,614
Wronglock(1,20)	7,391	0.78	97	562	2328	99
Wronglock(10,1)	$2,330,993^*$	1.00	795	4,848	26,070	834
Wronglock(20,1)	$2,056,532^*$	1.00	3,176	32,484	163,642	6,282

Table 2. Comparing the performance of default order guided search (GDS) and randomized guided search (Randomized-GDS) using the Estes model checker.

Model	GDS	Randomized-GDS				
		PED	Minimum	Mean	Maximum	95% CI
FSM Heuristic						
Barbershop(5)	132,376	1.00	13,917	59,496	154,473	5,948
Barbershop(9)	492,166	0.59	61,732	785,698	2,003,928	118,996
Barbershop(11)	$1,292,835^*$	0.15	381,808	813,644	1,247,461	157,172
e-fca Heuristic						
Barbershop(5)	814	1.00	921	1,012	1,308	13
Barbershop(9)	1,070	1.00	1,543	1,692	1,918	18
Barbershop(11)	1,196	1.00	1,939	2,243	2,671	27
Barbershop(20)	1,767	1.00	5,099	6,319	8,439	131
Barbershop(25)	2,086	1.00	7,654	9,873	12,657	233
EFSM Heuristic						
Barbershop(5)	21,706	1.00	4,950	19,849	67,875	1,853
Barbershop(9)	17,537	0.65	94,357	816,848	1,999,595	129,344
Barbershop(11)	30,256	0.06	293,893	701,278	1,181,985	412,829

tics in Estes from the study in JPF is that the randomized GDS trials and guided search using default search order are allocated 2 GB of RAM. The performance of the FSM distance heuristic function improves with the new randomized GDS algorithm as seen in Table 2. In the `Barbershop(11)` model, the default order guided search does not find an error in over 1.2 million states while the new randomized GDS algorithm explores only $813,644$ states—on average—in 15 error discovering trials.

It is interesting to note that for some models, the default order guided search outperforms the new randomized GDS algorithm using the EFSM and e-FCA distance heuristics. For example, in the `Barbershop(20)` model, 1767 states are generated with guided search while the minimum number of states generated by the randomized GDS algorithm is 5099. The examples where default order guided search outperforms the new randomized GDS algorithm support the hypothesis presented in [7] that certain reported performance gains of directed model checking techniques can potentially be an artifact of the default order implemented by the model checker rather than the technique itself.

This empirical study shows—on average—that the new randomized GDS algorithm is a better search technique than a default order guided search with no randomization. As a side note, we omit the results on the n-best algorithm in [23] and JPF's random choice generator because they are not competitive with the new randomized GDS algorithm. For the remainder of this paper, we use randomized GDS to refer to our new randomized GDS algorithm. The next section shows in another empirical study that with the correct heuristic, ran-

domized GDS performs well in the models where randomized DFS is unable to find an error. We refer to these models as *hard* [32].

4 Evaluation

Randomized DFS serves as a good standard for comparison when we evaluate the performance gains of randomized GDS [32]. Randomized GDS and randomized DFS both effectively control for the default search of the model checker implementation which makes them well-suited for comparison. Also, when evaluating the performance of a new heuristic, it is sometimes hard to find another heuristic that is designed to work on the same class of programs or properties. Randomized DFS serves as an ideal comparison technique to evaluate the performance of such heuristics. It also provides a tighter lower bound on performance than say a metric based on stateless random walk, [32], and is a significant bar to overcome when showing performance gains in stateful techniques such as randomized GDS.

We design an empirical study to compare the performance of existing heuristics, using randomized GDS, to randomized DFS implemented by JPF. Like the previous study, we run 100 trials of randomized GDS for each model and an equal number of randomized DFS trials. We bound the execution time at 1 hour for each trial. In our initial experiments, the size of the frontier, states in the priority queue, increases rapidly in randomized GDS trials which causes the searches to run out of memory in JPF before reaching the specified time bound. To overcome this issue, we bound the size of the queue in JPF at 100,000 states. This allows randomized GDS trials to successfully run for an hour in JPF without exhausting the available memory. Bounding the size of the queue turns the complete search into a partial search; however, guided search aims to find a counter-example efficiently rather than to do an exhaustive proof. An earlier study, [16], and our experiments show that bounding the size of the queue does not affect, in general, the number of randomized GDS trials that discover an error. The system configuration used to conduct this empirical study is the same as described in the previous section.

We record and normalize values of five different metrics in the randomized GDS and randomized DFS trials to study the performance gains of randomized GDS over randomized DFS. We measure the path error density, number of states generated, time taken before error discovery, length of the counter-example, and total memory utilized for each of the search trials. Recall that the path error density is the ratio of the error discovering trials over the total number of trials executed. We measure the minimum, mean, and maximum values for all metrics, except path error density, generated during the error discovering trials since the randomization generates different results in each trial. The minimum, mean, and maximum values generated by the search trials are normalized between 0.00 and 1.00 for each metric. Here is an explanation of the normalization process for states generated: the smallest number of states generated among the trials of both search techniques, for a given model, is mapped

Table 3. Comparing the average values generated in error discovering trials of randomized guided search (RGDS), using the Prefer-Thread heuristic, and randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Accountsubtype(1,1)	1.00	1.00	0.98	0.58	0.58	0.68	0.37	0.45	0.62	0.60
Accountsubtype(2,2)	1.00	1.00	1.00	0.59	0.99	0.60	0.42	0.36	0.99	0.37
Wronglock(10,1)	1.00	1.00	1.00	0.79	0.89	0.70	0.34	0.65	0.98	0.78
Wronglock(1,1)	1.00	1.00	0.89	0.52	0.55	0.94	0.70	0.49	0.58	0.56
Wronglock(1,10)	1.00	0.97	0.47	0.98	0.45	0.98	0.57	0.53	0.90	0.93
Twostage(1,1)	1.00	1.00	0.83	0.48	0.66	0.83	0.39	0.54	0.40	0.67
Twostage(2,5)	1.00	0.96	0.52	0.91	0.54	0.94	0.44	0.59	0.39	0.78
Twostage(6,1)	1.00	0.98	0.60	0.87	0.62	0.92	0.31	0.64	0.87	0.63
Reorder(5,1)	1.00	1.00	0.34	0.72	0.34	0.83	0.45	0.75	0.44	0.79
Reorder(8,1)	1.00	0.89	0.36	0.84	0.40	0.92	0.41	0.72	0.89	0.61
ProdCons(1,16,4)	0.67	0.87	1.00	0.88	0.99	0.85	0.55	0.72	1.00	0.67
Twostage(7,1)	0.41	0.73	0.42	0.76	0.42	0.89	0.17	0.58	0.97	0.53
Wronglock(1,20)	0.28	0.81	1.00	0.99	1.00	0.99	0.50	0.62	1.00	0.99
Reorder(9,1)	0.06	0.57	0.31	0.75	0.16	0.87	0.10	0.74	0.99	0.48
Twostage(8,1)	0.04	0.57	0.70	0.70	0.40	0.74	0.01	0.50	0.99	0.43
Reorder(10,1)	0.00	0.34	0.00	0.63	0.00	0.70	0.00	0.51	0.00	0.38

to the value of 1.00; similarly, the largest number of states generated among the trials is mapped to the value of 0.00. All other values for states generated, in the given model, are normalized between these two values. The values are normalized to the maximum or minimum values since these represent the extremes in the observed performance across several trials. The normalization process is conducted separately for each metric in a model. Intuitively, values close to 1.00 indicate good performance for a given metric while values close to 0.00 indicate the opposite. The normalization technique helps us in better understanding and visualizing the performance of the heuristic in different models because it puts all metrics on the same scale and graph across both search techniques.

The prefer-thread heuristic, using randomized GDS, performs well in the models shown in Table 3. Please note that this table omits the data for the minimum and maximum values across our several metrics. Table 3 only presents the average values that have been normalized. The values given in Table 3 are as follows: path error density (`PED`), number of states (`States`), time taken (`Time`), length of counter-example (`Trace`), and memory utilized (`Memory`) measured in error discovering trials of randomized GDS and randomized DFS. In a large number of models, the path error density is the same, 1.00, for both randomized DFS and randomized GDS. In models where randomized DFS has a path error density of 1.00, finding an error is not difficult, and the results on these models do not convey much information on the effectiveness of the heuristic.

To overcome some of the weakness in the benchmarks, our study uses hard models generated in [32] to evaluate the true effectiveness of the heuristic, which are the last six entries in Table 3. For example, in the `Wronglock(1,20)` model, the measured path error density of randomized DFS is 0.28 while the path error density of the randomized GDS is dramatically higher at 0.81. The average values for states, time, and memory are close to 1.00 for both search techniques in the `Wronglock(1,20)` model; however, the average length of the counter-example for randomized GDS is smaller than the average length of the counter-example recorded from the randomized DFS trials. In understanding the length of a counter-example, values closer to 1.00 depict a shorter counter-example while values close to 0.00 indicate a longer counter-example. There are other models like `Reorder(9,1)`, `Twostage(8,1)`, and `Reorder(10,1)` where randomized GDS improves over randomized DFS.

The high path error density of randomized GDS in models where randomized DFS struggles to find an error makes a compelling argument for the use of the heuristic in the given models. The results in Table 3 show that randomized GDS, using the prefer-thread heuristic, successfully overcomes the lower bound on the performance set by randomized DFS in the given models.

In Fig. 3 we visualize the comparative performance of randomized DFS and randomized GDS using the prefer-thread heuristic for the models shown in Table 3. The minimum, mean, and maximum values for all the different metrics and models are aggregated in Fig. 3(a). The different edges along the graph show which search technique generates the best and worst boundary values. The points in the graph along the axis where $x = 0$ show all the worst values that are contributed by randomized DFS for the measured metrics while the points along the axis where $y = 0$ show all the worst values generated by randomized GDS. Similarly, points along $x = 1$ represent the best values contributed by randomized DFS while points along $y = 1$ represent the best values contributed by randomized GDS. The points above the dashed diagonal line in Fig. 3(a) show the values of the metrics where randomized GDS improves over randomized DFS. In general, there is a high density of points above the diagonal that show for the given set of models, it is more effective to use randomized GDS, with the prefer-thread heuristic, over randomized DFS. There is also a high density of points in the upper right corner of the graph. These points represent the values where both randomized GDS and randomized DFS perform well and do not help us in evaluating the true effectiveness of the search and heuristic over randomized DFS. We now look at each of the metrics separately to understand the areas in which randomized GDS scores over randomized DFS.

There are three metrics where randomized GDS clearly outperforms randomized DFS in the benchmark suite using the prefer-thread heuristic. These three metrics are the path error density, length of the counter-example, and time taken before error discovery as shown in Fig. 3(b), (c), and (d) respectively. The points in the upper right corner of the graph in Fig. 3(b) show that in all trials, both search techniques are equally successful in finding the error; however, points that are above the dashed diagonal line show that a larger number of random-

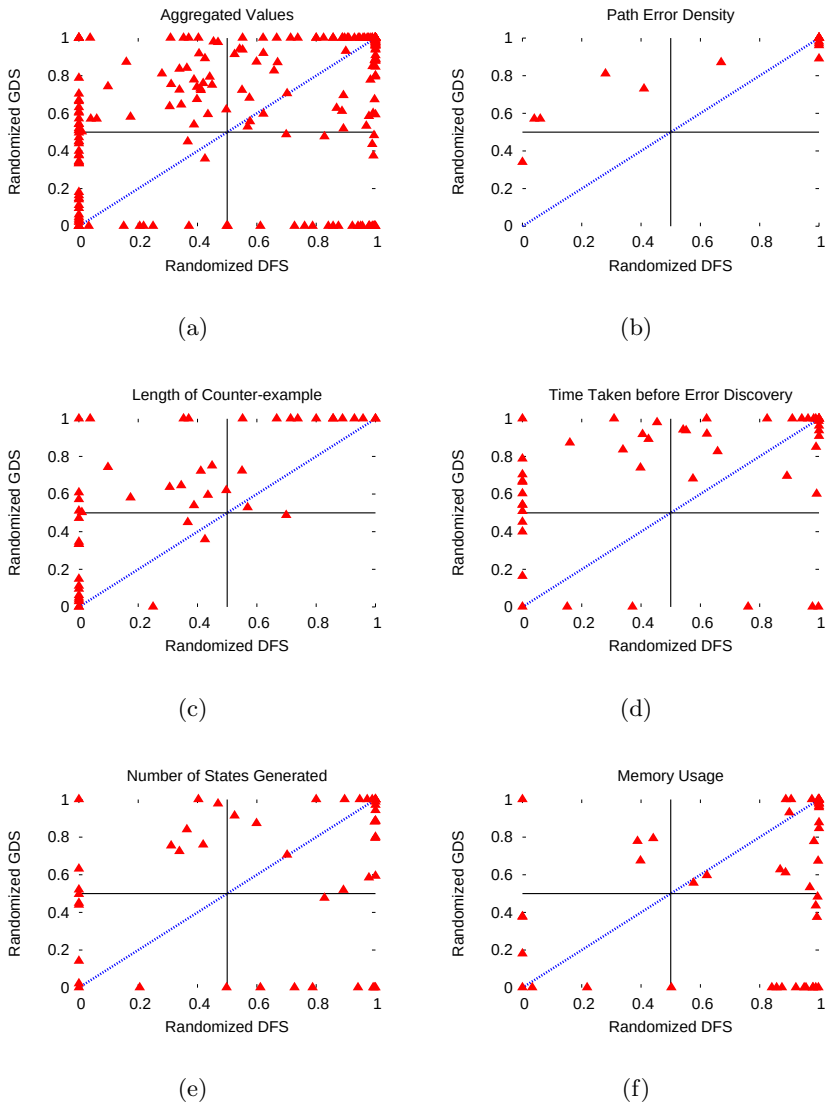

Fig. 3. Visualizing the normalized minimum, mean, and maximum values of different metrics comparing randomized GDS, using the Prefer-Threads heuristic, to randomized DFS. (a) An aggregation of all values for the different metrics. (b) Values comparing path error density. (c) Values comparing length of counter-example. (d) Values comparing time taken before error discovery. (e) Values comparing number of states generated. (f) Values comparing memory usage.

Table 4. Comparison of results using the Most-Blocked Heuristic with a randomized guided search (RGDS) to results from randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Clean(1,1,12)	1.00	1.00	0.09	0.59	0.52	0.87	0.34	0.25	0.42	0.65
Piper(2,4,4)	1.00	1.00	0.96	0.65	0.96	0.63	0.60	0.85	0.94	0.25
Piper(2,8,4)	0.96	0.00	0.92	0.00	0.92	0.00	0.52	0.00	0.47	0.00
Clean(10,10,1)	0.96	0.00	0.95	0.00	0.96	0.00	0.37	0.00	0.85	0.00
Piper(2,16,8)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5. Comparison of results using the Interleaving Heuristic with a randomized guided search (RGDS) to results from randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Airline(6,1)	1.00	1.00	0.75	0.99	0.74	0.99	0.22	0.62	0.53	0.90
Airline(6,2)	1.00	1.00	0.96	1.00	0.95	1.00	0.25	0.60	0.89	0.97
Raxextended(4,3)	1.00	1.00	0.96	0.99	0.96	1.00	0.67	0.99	0.87	0.96
Airline(20,4)	0.03	0.00	0.55	0.00	0.59	0.00	0.47	0.00	0.39	0.00
Airline(20,3)	0.01	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
Airline(20,2)	0.01	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00

ized GDS trials find an error in models where only a small number of randomized DFS trials find an error. The high path error density of randomized GDS is a very compelling measure that depicts the improvement of randomized GDS over randomized DFS. Randomized GDS also performs extremely well in generating shorter counter-examples. The high density of points above the diagonal in Fig. 3(c) indicates that randomized GDS has dramatically shorter counter-examples compared to randomized DFS across all the models in test. Similarly, the distribution of points in Fig. 3(d) indicates that randomized GDS takes less time to find an error when compared to randomized DFS.

In Fig. 3(e), it is hard to discern which search technique performs better in generating fewer number of states before error discovery; however, the randomized DFS clearly outperforms randomized GDS in the amount of memory utilized as shown in Fig. 3(f). Randomized GDS maintains the frontier of states that need to be explored. The increasing frontier size, however, has a dramatic impact on the memory usage. The unbounded priority queue in JPF causes a serious explosion in memory usage while executing the randomized GDS. In fact, as mentioned earlier, we restrict the size of the priority queue to only 100,000 states so that 7 GB of RAM is not exhausted before reaching the specified time bound. Overall, across the different metrics, randomized GDS using the preferthread heuristic improves performance over randomized DFS by effectively finding counter-examples and generating shorter counter-examples.

Table 6. Comparison of results using the Choose-Free Heuristic with a randomized guided search (GDS) to results from randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Deos(true)	1.00	1.00	0.72	0.97	0.56	0.96	0.36	0.95	0.60	0.92
Replicated(5,2)	0.97	0.00	0.81	0.00	0.87	0.00	0.57	0.00	0.88	0.00
RWNoExpChk	0.77	1.00	0.97	0.72	0.72	0.55	0.75	0.99	0.94	0.69

We present results for the most-blocked, interleaving, and choose-free heuristics in Table 4, Table 5, and Table 6 respectively. These heuristics do not perform well on the class of models for which they are designed, and the comparison with randomized DFS makes these heuristics even less appealing in our benchmarks. For example, the randomized DFS path error density for `Piper(2,8,4)` model is 0.96 while the path error density of randomized GDS using the most-blocked heuristic as seen in Table 4 is 0.00. Similar behavior is seen for the model `Clean(10,10,1)`. The choose-free, most-blocked, and interleaving heuristics do not overcome the randomized DFS lower bound and are not effective in generating counter-examples for models in the tables. The sub-par performance of these heuristics argues a greater need to identify models where they are effective.

The results in this section indicate that given the correct heuristic for a set of models, randomized GDS is effective in finding errors where randomized DFS struggles. It is also important to note that better error discovery, shorter counter-examples, and reduced error discovery time in randomized GDS comes at the cost of increased memory usage due to the large search frontier.

5 Conclusions and Future Work

This paper presents a new randomized GDS algorithm that completely shuffles states in the priority queue with equal heuristic rankings. The algorithm is easily implemented, efficient, and has low overhead in terms of memory and time. We show that full randomization of the guided search improves the effectiveness of the search over the regular guided search. To evaluate the performance of randomized GDS using a particular heuristic, we compare it with randomized DFS because randomized DFS creates a lower bound for establishing performance gains in directed model checking. Also, when the heuristic is correctly matched to the models and properties, the new randomized GDS algorithm outperforms randomized DFS in both the effectiveness of the search in finding counter-examples and the length of the counter-examples. The approach is timely given the recent explosion in computation resources and is easily distributed to several computation nodes to improve the likelihood of error discovery.

There is a need to explore other avenues for combining randomization and directed model checking. For example, can we use randomization to balance exploring new parts of the behavior space and use heuristics to exploit the infor-

mation available about the model? Also, as we develop heuristics appropriate for use in a randomized GDS algorithm, there is a need to understand the intended problem domain for the heuristic. In other words, we need to characterize heuristics in terms of the models for which they are expected to be effective. Without this characterization, it is not obvious which heuristic best fits a given property and model. There also a need to define language and metrics to characterize heuristics for their intended problem domains. An interesting avenue of research is to use something similar to the *"Patterns"* categorization for specifications [5].

6 Acknowledgments

We thank Matt Dwyer and Suzette Person at the University of Nebraska for sharing with us the models presented in [7] and the discussions of their work on the quality of models. We also thank Shmuel Ur for providing us access to the models developed at the IBM Research Center in Haifa. We finally thank Ira and Mary Lou Fulton for their generous donations to the BYU Supercomputing laboratory which made it possible for us to run the extensive analysis presented in this paper.

References

1. J. Barnat, L. Brim, and J. St. Distributed LTL model checking in SPIN. In *Proceedings of the 8th International SPIN workshop on Model Checking of Software*, pages 200–216. Springer-Verlag New York, Inc., 2001.
2. J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.
3. L. Brim, I. Cerna, P. Moravec, and J. Simsa. How to order vertices for distributed LTL model-checking based on accepting predcessors. *Electronic Notes in Theoretical Computer Science*, 135(2):3–18, February 2006.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
5. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.
6. M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
7. M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 92–104, New York, NY, USA, 2006. ACM Press.

8. S. Edelkamp and S. Jabar. Large-scale directed model checking LTL. In A. Valmari, editor, *13th International Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.
9. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 7th International SPIN Workshop*, number 2057 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
10. S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
11. S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.
12. Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurrency and Computation: Practice & Experience*, 19(3):267–279, 2007.
13. Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, page 266a, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
14. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.
15. Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
16. A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21, July 2002.
17. P. Haslum. Model checking by random walk. In *Proceedings of ECSEL Workshop*, 1999.
18. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
19. G. J. Holzmann. The design of a distributed model checking algorithm SPIN. FMCAD 2006 Invited Presentation, November 2006.
20. C. P. Inggs and H. Barringer. CTL* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, 2006.
21. S. Jabbar and S. Edelkamp. Parallel external directed model checker with linear I/O. In E. A. Emerson and K. S. Namjoshi, editors, *Seventh International Conference on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 237–251, 2006.
22. M. Jones, E. Mercer, T. Bao, R. Kumar, and P. Lamborn. Benchmarking explicit state parallel model checkers. In *2nd International Workshop on Parallel and Distributed Methods in Verification*, 2003.
23. M. D. Jones and E. Mercer. Explicit state model checking with Hopper. In *International SPIN Workshop on Software Model Checking (SPIN'04)*, number 2989 in LNCS, pages 146–150, Barcelona, Spain, March 2004. Springer.
24. M. D. Jones and J. Sorber. Parallel search for LTL violations. *Software Tools for Technology Transfer*, 7(1):31–42, 2005.
25. T. Krazit. Intel pledges 80 cores in five years. CNET News.com, September 2006.

26. E. G. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265, San Francisco, USA, August 2005. Springer.
27. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):34–48, November 2003.
28. R. Pelanek, T. Hanzl, I. Cerna, and L. Brim. Enhancing random walk state space exploration. In *FMICS '05: Proceedings of the 10th International Workshop on Formal methods for industrial critical systems*, pages 98–105, New York, NY, USA, 2005. ACM Press.
29. J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.
30. N. Rungta and E. G. Mercer. A context-sensitive structural heuristic for guided search model checking. In *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 410–413, Long Beach, California, USA, November 2005.
31. N. Rungta and E. G. Mercer. An improved distance heuristic function for directed software model checking. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 60–67, Washington, DC, USA, 2006. IEEE Computer Society.
32. N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. Technical Report SMC-BYU-0107, Brigham Young University, Department of Computer Science, 2007.
33. K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. *Fundamenta Informaticae*, 70(1-2):111–126, 2006.
34. H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proceedings of Workshop on Parallel and Distributed Model Checking*, 2003.
35. U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. In O. Grumburg, editor, *Computer-Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267, Haifa, Israel, June 1997. Springer-Verlag.
36. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder: Second generation of a Java model checker. In G. Gopalakrishnan, editor, *Proceedings of the Workshop on Advances in Verification (WAVE'00)*, July 2000.
37. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *35th Design Automation Conference (DAC98)*, pages 599–604, 1998.