Search-Based Test Generation for Robotic Motion Planning Algorithms

Amanda Baughan
University of Washington
baughan@cs.washington.edu

Nathan Hatch
University of Washington
nhatch2@cs.washington.edu

Vinitha Ranganeni
University of Washington
vinitha@cs.washington.edu

Boling Yang
University of Washington
bolingy@cs.washington.edu

Abstract

Testing robot motion planning algorithms is a repetitive and time-consuming manual task for roboticists. This work introduces and evaluates a tool that automatically verifies whether a planner has been implemented correctly. We formulate the planner verification problem as an optimization task over obstacle configurations, which we solve using a gradient-free optimization algorithm. Our experimental evaluation shows that compared to human verification, this approach achieves comparable accuracy and requires significantly less time. Simple random testing performs similarly well. We discuss the relevance and implications of our findings for both the software engineering and robotics research communities.

Keywords

Search-Based Planning, Testing, Robotics, Motion Planning, Dynamic Program Analysis, Random Testing

ACM Reference Format:
Amanda Baughan, Nathan Hatch, Vinitha Ranganeni, and Boling Yang. 2021. Search-Based Test Generation for Robotic Motion Planning Algorithms. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

One reason we do not currently see robots operating alongside humans in everyday life is because they cannot effectively navigate complex environments. The area of robotics that attempts to address this challenge is called *motion planning*. A motion planning algorithm (planner) seeks to find a path from one configuration or position to another that avoids colliding with obstacles. Planning algorithms can be implemented using a variety of approaches, however, our solution is targeted towards search-based planning in discrete space (see 2.1).

When implementing a planner, many different kinds of bugs can arise. A large part of the implementation effort goes towards testing the planner to verify that it is working as expected. Some examples of ways that a planner could fail are:

- **Heuristic values overflow**. Some planners use a *heuristic function* to guide the planner towards the goal. Heuristic values are often represented as integers. Heuristic (integer) overflow is a common issue, especially when planning in high-dimensional spaces, since the number of states expanded and their associated values can increase exponentially with an increase in dimensionality.
- **Infinite exploration**. If a planner continually searches over duplicated states, it may never terminate, or it may run out of memory and crash.
- **Unable to find solution**. If a planner neglects to check whether the start and goal state are within the world boundaries, it may spend a long time looking for a solution that does not exist.

Currently, the typical testing strategy is a manual *ad hoc* implementation of undirected random testing: a roboticist will choose random start and goal configurations to input into the planner. However, this is not guaranteed to catch any of the failure cases listed above. Often these failures occur in challenging scenarios where the heuristic guides the search towards a region often referred to as a "local minimum." In these regions the search ceases to progress towards the goal or progresses extremely slowly. Manually constructing scenarios that contain local minima is tedious and time-consuming.

In this work we automatically generate challenging scenarios with local minima using evolutionary search techniques in order to catch several different types of bugs. Our experimental evaluation shows that compared to human verification, this approach achieves comparable accuracy and requires significantly less time. However, we found that random testing performs just as well as evolutionary search. This finding extends and confirms prior works in software testing research [15], in which random automated testing has been found to achieve similar results to evolutionary testing.

1.1 Motivating Example

Imagine a team of roboticists are trying to plan a footstep path (i.e. a plan for a bipedal robot's feet). Each foot is represented by a 3-dimensional configuration (x, y, θ) where (x, y) is the position and θ is the orientation of the foot. This requires planning in a 6-dimensional space. When planning in a high-dimensional space there is much room for implementation error, and creating test

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

cases that effectively expose errors is almost impossible with the current manual, undirected random testing approach. In this video, the blue and purple rectangles represent the planner's search for a path. The planner spends a lot of time trying to see if it can fit through the gap between the chair and the kitchen island (a local minimum). It exhaustively searches all possible states inside this gap, and then eventually plans around the island. If the planner was not implemented correctly, it could have run out of memory and crashed during this computation, or failed to escape from the local minimum. The gap in this example was manually engineered to be a difficult distance to plan a path through, and developing this test case was highly time-intensive.

2 Background and Related Work

2.1 Search-Based Motion Planning in Robotics

Search-based planning algorithms (for example, A* and its associated algorithms [6, 12]) use graph search methods to find paths in a discrete representation of the world. A prerequisite to planning motion discretely is a finite set of actions that can be applied to a finite set of states. The solution consists of an appropriate sequence of actions [10]. Many search-based planning approaches use a heuristic function to guide the planner's search towards the goal. The algorithm is considered complete if its heuristic function is *admissible* and *consistent*. *Admissible* means the heuristic function never overestimates the cost of reaching the goal; *consistent* means the heuristic function's estimated "cost" or distance left to traverse from any given vertex is always less than or equal to the estimated distance from any neighboring vertex to the goal, plus the step cost of reaching that neighbor [10]. One benefit of search-based planning is that when successful, the solution is guaranteed to be optimal. However, it is computationally unwieldy when working in highly dimensional spaces. Therefore, search-based planning typically is employed only in low-dimensional spaces.

2.2 Testing in Software Engineering

2.2.1 Search-based Testing Search-based testing refers to applying search-based optimization to the generation of software test data or test cases. The objective function in search-based optimization algorithms is called the fitness function [8]. The fitness value is a numerical value that expresses the performance of candidate solutions, with regard to the current optimal candidate solution to allow for comparison, with the goal of satisfying the test criterion [1, 14]. The fitness function guides search in choosing candidate solutions for reproduction, gradually improving fitness values with each generation until a solution is found. The notion of fitness is critical to the application of search-based testing algorithms: their success depends on the use of a fitness function that changes neither too rapidly nor too slowly within the design parameters of the optimization problem [1]. Different fitness functions can be defined to capture different test objectives, allowing the same search-based optimization strategy to be applied to different test data generation scenarios [1, 8, 16]. It has been shown that search-based testing outperforms random testing in many cases, as random testing doesn't provide full coverage [8, 16]. However, one of the drawbacks of search-based techniques is that they can become stuck in local optima and perform poorly if the search landscape or fitness function offers no guidance [13]. For this reason, metaheuristic search methods such as evolutionary algorithms are often more practical and generalizable than neighborhood search methods such as hill climbing [1]. In evolutionary tests, an initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions, and introducing small random changes [1, 8]. Evolutionary tests have been shown to be robust and suitable for the solution of different test tasks [1, 8, 11, 13, 16]. The *covariance matrix adaptation evolution strategy* (CMA-ES) is one such example of an evolutionary algorithm [7].

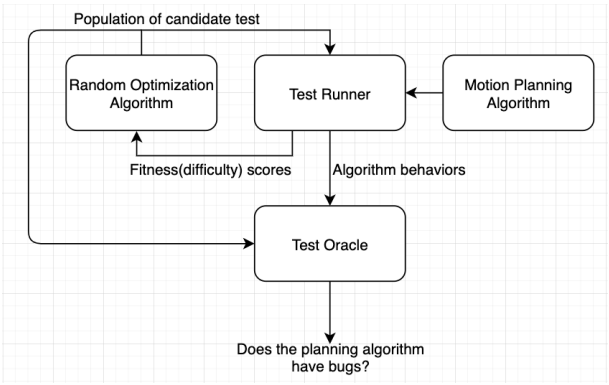


Figure 1: Flow chart of our approach

2.2.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) In CMA-ES, candidate solutions are sampled from a multivariate normal distribution in $\mathbb{R}^n$. At each iteration, the mean and covariance matrix of this distribution are updated based on the ranking between candidate solutions. Neither derivatives nor the fitness function values themselves are required by the method [7]. It is distinct from other approaches such as the Cross-Entropy Method in that it estimates the covariance matrix by maximizing the likelihood of successful search *steps* rather than solution *points* [5, 7]. CMA-ES has been empirically successful in hundreds of applications and is considered to be particularly useful with non-convex, ill-conditioned, multi-modal or noisy objective functions [3, 4, 9].

2.2.3 Test Oracles A test oracle is a source of information about whether the output of a program is correct or not. A test oracle might specify correct output for all possible inputs or only one specific input. It also may not specify actual output values, but only the constraints on them [2].

3 Approach

Our approach employs CMA-ES [7], a gradient-free optimization technique, to search the space of test environments and generate challenging test cases for search-based robotics motion planning.

Each test environment is represented by a vector of real numbers, which represent *obstacle configurations*. We choose a fitness function that seeks *obstacle configurations* that are difficult to generate manually and most likely to cause failures. The fitness value of the environment is evaluated by running the motion planning

Algorithm 1 A-Star Algorithm

```
1:  function A*(q_g, q_u)
2:      if !is_valid(q_g) || !is_valid(q_u) then
3:          return NIL
4:      Q = ∅
5:      g[q_u] ← 0
6:      Q.add_with_priority(q_u, 0)
7:      while Q ≠ ∅ do
8:          q_v ← Q.extract_min()
9:          if dist(q_v, q_g) < ϵ then
10:             return extract_path(q_u, q_v)
11:         V_succ ← succ(q_v)
12:         for q'_v ∈ V_succ do
13:             if !is_valid(q'_v) then
14:                 continue
15:             alt ← g[q_v] + length(q_v, q'_v) + h(q'_v)
16:             if g[q'_v] = NIL || alt < g[q'_v] then
17:                 g[q'_v] ← alt
18:                 Q.add_or_update_priority(q'_v, g[q'_v])
19:     return extract_path(q_u, q_v)
```

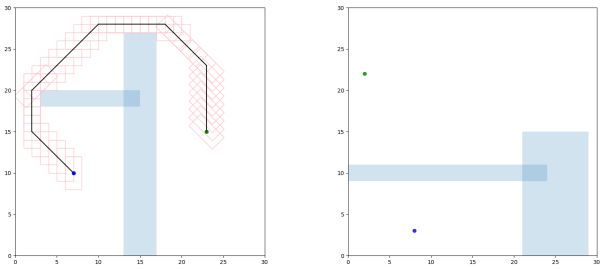
algorithm and collecting several data points about the run, such as the number of *nodes* expanded in the search tree and the wall clock *time*.[1] Once we have the fitness values, we apply the CMA-ES optimization update [7] to generate a new set of candidate environments with *obstacle configurations* designed to be more difficult than the previous set. We continue iterating until the motion planning algorithm fails or until the difficulty of the environments seems to plateau (as measured by our fitness function).

As discussed in Section 2.2.3, automatic detection of certain kinds of bugs requires a test oracle. For the motion planning application, given an *obstacle configuration*, an oracle should specify whether the planning problem is feasible, and if so, what is the length of the optimal path. This essentially amounts to solving the original planning problem. In many cases, it is unrealistic to assume access to such an oracle. Hence, for our experiments, we designed a set of *partial oracles*, each intended to catch one specific kind of bug. If for a given test input, the planner triggers any of these test oracles, then we have found a failing test case. Figure 1 shows a diagram of our approach architecture.

4 Experimental Methodology

Our experiments are based on a Python implementation of A* (see Alg. 1). A* computes the shortest path between the given start configuration and goal region, q_u and q_g respectively. First the algorithm checks if the given configurations are not in collision and in the bounds of the environments (Line 2). Then it initializes the priority queue Q (Line 4) and sets the g-value (cost to get to the node) of q_u in the g-value (cost-to-come) map (Line 5). While the priority queue is not empty, A* pops a node q_v off the priority queue (Line 8). If the distance between the node popped and the goal configuration, q_v, is less than some ϵ representing the radius of the goal region, the

[1]For more details, see Section 4.



(a) (b)

Figure 2: (a) Example of a possible planning scenario, with solution superimposed. Units are centimeters. (b) Example of infeasible planning scenario

shortest path between q_u and q_v is returned (Line 9).[2] Otherwise, we find the successors of q_v and for each successor q'_v, check that it is valid (Line 13), compute its cost (Line 15), and add or update its priority in the queue (Line 18). If all the nodes are expanded, A* returns the partial path to the goal (Line 19).

We refer to this as the `ref` (reference) planning algorithm. We create five buggy implementations of A* by manually introducing bugs into the reference implementation at various places, specified by the line number in the pseudo code, detailed below.

- `Bug1`: The ϵ (radius of the goal region) in the goal termination criteria is too small to be achieved while calculating in discrete space (Line 9).
- `Bug2`: Planner extracts the path to the goal region q_g instead of configuration q_v in the goal region (Line 10, see footnote 2).
- `Bug3`: Use L_1 distance as the A* heuristic instead of L_2 distance, so that it is no longer consistent (Line 15).
- `Bug4`: Multiply all distance and heuristic values by 2^{30} to induce numerical overflow (Line 15).
- `Bug5`: Don't check whether a discovered node has already been expanded (Line 16).

For each of these bugs, we implemented a corresponding oracle as follows:

- *False negative* (`Bug1`): The algorithm says that there is no feasible plan even when there are no obstacles.
- *Invalid plan* (`Bug2`): The algorithm reports success, but cannot extract a valid path to the goal region.
- *Inconsistent heuristic* (`Bug3`): The heuristic values along the solution path do not obey the triangle inequality. Note that this oracle requires access to additional details of the planning algorithm, namely the heuristic values along the solution path.
- *Integer overflow* (`Bug4`): The processor detects integer overflow while the algorithm is running.

[2]It is difficult for the planner to find a path to an exact goal configuration due to information loss when discretizing and the complexity of the state space. Thus, it is common to plan a path to a goal region (i.e. a set of states that are within some epsilon of the desired goal state)

	random	human	CMA
ref	1.0	1.0	1.0
bug1	1.0	1.0	1.0
bug2	1.0	1.0	1.0
bug3	1.0	1.0	1.0
bug4	1.0	0.5	1.0
bug5	1.0	1.0	1.0

Table 1: Fraction of trials in which the validation algorithm correctly identified whether the implementation was buggy

	random	human	CMA
ref	169.8 ± 3.7	480 ± 350	271 ± 38
bug1	8.02 ± 0.22	14.0 ± 3.2	8.54 ± 0.28
bug2	0.69 ± 0.25	40. ± 29	0.88 ± 0.55
bug3	1.11 ± 0.52	143 ± 55	1.7 ± 1.3
bug4	9.4 ± 6.2	115 ± 58	7.6 ± 6.3
bug5	12.7 ± 3.3	21.2 ± 2.5	15.36 ± 0.12

Table 2: Computation time in seconds (average ± standard deviation)

- *Failure to terminate* (Bug5): The planner does not complete within fifteen seconds. In general, to avoid false positives from this oracle, the practitioner should know an upper bound on the time required for the planner to terminate. For the planning problems described below, typical worst-case completion time of an accurate planner was eight seconds.

We test these planning algorithms in a simulated 2D environment that is $0.3m \times 0.3m$ and has a grid resolution of $0.01m$. The robot is a $0.04m \times 0.02m$ rectangular robot whose pose consists of 2D location (x, y) and rotation θ. For a given robot pose $p = (x, y, \theta)$, the action space (i.e. set of possible actions the robot can apply) is $\{(x \pm 0.01, y, \theta),\ (x, y \pm 0.01, \theta),\ (x \pm 0.01, y \pm 0.01, \theta)\},\ (x, y, \theta \pm \frac{\pi}{4})\}$. The obstacles are the world boundaries and a collection of K rectangles $r_1, \ldots, r_K$. Our experiments used $K = 2$. Each rectangle has parameters $r_k = (x_k, y_k, \ell_k, w_k)$ comprising 2D location, length, and width. The start pose p_S and goal pose p_G each have three parameters $(x, y, \theta)_{\{S,G\}}$ comprising 2D location and orientation. The goal region consists of poses (x, y, θ) such that $|x - x_G| \leq 0.005$, $|y - y_G| \leq 0.005$, and $|\theta - \theta_G| \leq 0.5$, where angles are given in radians. These $4K + 3$ parameters describe all possible testing environments.[3] The planning problem is to find the shortest sequence of poses $p_0, \ldots, p_N$ such that $p_0 = p_S$, p_N is within the goal region, and none of the poses are in collision with any of the obstacles. A screenshot of one possible testing environment is shown in Figure 2a. Note that in many of these environments, the planning problem is infeasible (see Figure 2b).

4.1 Evaluation

We compare three different approaches to verifying these A^* implementations.

Our first baseline, human, is manual verification by humans. We give a human a random, blind planner implementation. They are tasked with either (1) identifying the bug and exhibiting a test case for which the algorithm fails, or (2) declaring that the implementation is correct. For each of these trials, we collect the following data:

- Whether they correctly identified whether the implementation was buggy
- How much wall-clock time elapsed before they made their decision

[3]Catching bug1 is a special case, because the test oracle requires running the planner on an environment with no obstacles. When collecting data about catching this bug, the validator knows that the planner implementation is either ref or bug1, and the test input consists only of the start and goal poses.

Each time a planning algorithm is executed, we collect the following data:

- How many nodes were expanded in the A* search tree
- Whether any of the test oracles detected a bug

Our second baseline, random, is an automatic algorithm that randomly generates test cases. Parameters were drawn from the distribution below:

- Obstacle length and width drawn from Unif($[0, 0.12]$)
- Obstacle, start, and goal location drawn from Unif($[0, 0.3]$)
- Start and goal orientation drawn from Unif($[0, 2\pi]$)

If the planning algorithm fails within some fixed number M of such tests, the random verification algorithm reports this test case; otherwise it declares the implementation correct. We used $M = 500$, because this roughly matched the number of tests run by the optimization-based algorithm (CMA, described below) when run on a bug-free implementation of the planner.

Finally, we test an optimization-based algorithm, CMA, as suggested in Section 3 above. For our fitness function, we use the number of nodes expanded in the A^* search tree. We use three random restarts, to avoid the problem of the algorithm being stuck at a poor initialization point, e.g. a local minimum or a region with no gradient signal. We terminate each random restart when the fitness value has not improved within the last five iterations.

For the automatic algorithms, due to randomness, for each combination of planner and verifier we report mean and standard deviation over five independent trials. For human testing, two computer science PhD students familiar with robotic motion planning performed blind validation tests until each bug had been encountered twice; we report mean and standard deviation over these two trials.

We consider a validator *successful* if it correctly identifies the whether each implementation is buggy on at least four of the five trials. We consider it *useful* if it finishes more quickly than human verification (human).

5 Results

Quantitative. Accuracy is shown in Table 1 and computation time is shown in Table 2. Both random and CMA achieve perfect accuracy using the given set of oracles. Hence, both methods are successful validators according to our criteria. One of the human trials failed to notice the heuristic overflow bug (bug4).

The random validator appears to be somewhat faster than CMA, although the difference is small except when running on the bug-free planner (ref). Both methods were faster than the human baseline, which makes both methods useful according to our criteria.

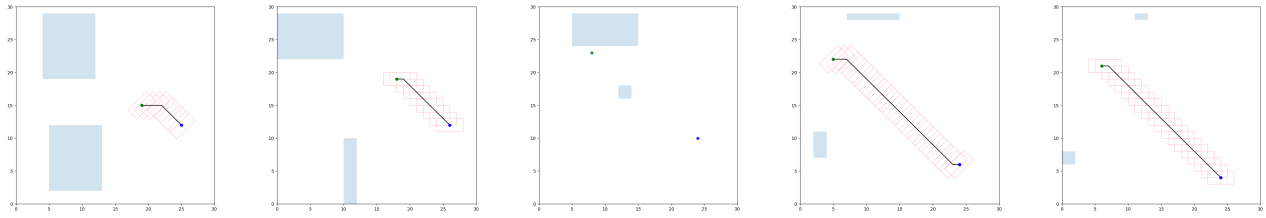


Figure 3: Five equally spaced tests from one of the CMA random restarts

The distribution of random test inputs described in the previous section results in many infeasible planning problems, usually because the start or the goal pose intersects one of the obstacles. Nonetheless, we observed that enough of the problems are feasible, and enough of the feasible problems are complicated enough, that all of the bugs that we considered are virtually guaranteed to occur within the first few dozen samples.

Qualitative. As shown in Figure 3, CMA tends to increase the number of node expansions by moving the start and goal farther from each other, moving obstacles out of the way if necessary to increase this distance. Hence, the optimized tests are not particularly interesting from a planning perspective. In addition, CMA requires many relatively similar samples in order to find a good update direction at each iteration, which means that it does not see a diverse set of test inputs as quickly as the random validator does. This may explain its relatively high computation time.

To investigate the internal behavior of the optimization-based approach, Figure 4 graphs learning curves for CMA on two of the implementations (`ref` and bug4). During validation of the reference implementation, the objective function increases gradually for most trials. This indicates that CMA generates test cases that require the planner to expand more nodes to come up with solutions compared to the previous iteration. While validating bug4, all trials stopped at a very early phase of the optimization process. We found that in our simple planning environments, most bugs are caught during the very first iteration of the CMA algorithm. This illustrates why `random` performs as well as CMA in our experiments. Figure 4c shows the learning curves on the same bug4 implementation in a larger 120×120 state-space. These longer and increasing learning curves suggest that more complex planning problems may be able to take better advantage of search-based testing.

5.1 Threats to Validity

An obvious threat to validity is that these buggy planner implementations did not occur "in the wild". Instead, we started from a valid implementation and manually introduced bugs. Future work may address this by investigating the commit history of a project with a previously buggy implementation of a planning algorithm. We attempted to mitigate this issue by testing a wide variety of potential bugs.

6 Conclusion

In this paper, we investigated the question of whether automated testing techniques could accelerate the testing process for robotic motion planning algorithms. By automating the process of finding challenging test cases, our goal was that researchers could focus on solving, rather than finding, those corner cases.

Our experiments show that given a high-quality set of test oracles and a reasonably good distribution for random test inputs, a high bug detection rate can be achieved simply by running the planner on a few dozen randomly sampled test inputs. By comparison, optimization-based automated testing yielded similarly accurate results, but at the cost of more time than random automated testing.

In more complicated and high-dimensional environments, our optimization-based approach to automated testing might outperform simple random evaluations [7]. For example, a humanoid robot has many degrees of freedom and operates in a 3D environment. In this case, it would likely be difficult to design a distribution of random test inputs that captures all of the desired edge case.

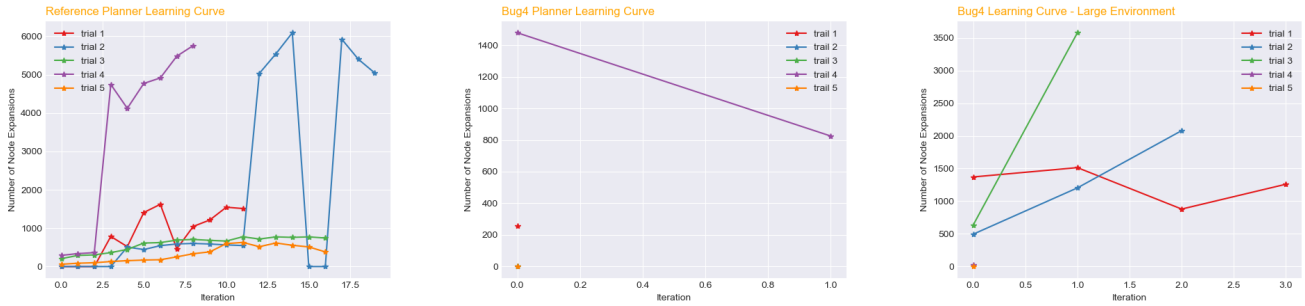
The performance of optimization-based testing might also improve with a different choice of fitness function. As noted above, the problems found by CMA when optimizing for the number of expanded nodes are not usually very interesting from a planning perspective. Optimizing for something like path length might encourage CMA to make more creative use of obstacles. Another choice that might work well for the heuristic overflow bug in particular (bug4) would be directly optimizing the maximum observed heuristic value.

Perhaps the most serious challenge to the usefulness of this approach is the difficulty of obtaining high-quality test oracles. We found that simple partial oracles like "the program should not crash" were not enough to detect common bugs in motion planning algorithms. In the end, each of our test oracles was designed to catch one specific bug. This unfortunately means that we could only catch bugs that we had anticipated. One way to get around this would be to have a reference implementation, known to be bug-free, which could function as a universal oracle. But in that case, it is hard to see why we would need to implement automated testing at all.

Overall, we find that with enough customization, automated testing is effective at catching bugs in robotic planning algorithms, but further research into applications in high-dimensional environments is needed to evaluate whether it will be useful in practice.

7 Code

Our code is made available at https://github.com/vinitha910/503-final-project.



(a) (b) (c)

Figure 4: (a) Learning curve of `CMA` on bug-free implementation in 30×30 grid environment, (b) on `bug4` in 30×30 grid environment, and (c) on `bug4` in 120×120 grid environment

References

[1] André Baresel, Harmen Sthamer, and Michael Schmidt. 2002. Fitness Function Design To Improve Evolutionary Structural Testing. 1329–1336.

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[3] P. Cerveri, A. Pedotti, and G. Ferrigno. 2004. Evolutionary optimization for robust hierarchical computation of the rotation centres of kinematic chains from reduced ranges of motion the lower spine case. *Journal of Biomechanics* 37, 12 (2004), 1881 – 1890. https://doi.org/10.1016/j.jbiomech.2004.02.032

[4] David Charypar, Kay W. Axhausen, and Kai Nagel. 2006. Implementing activity-based models: Accelerating the replanning process of agents using an evolution strategy. In *Conference On Issues In Behavioral Demand Modeling And The Evaluation Of Travel Time.*

[5] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. 2005. A tutorial on the cross-entropy method. *Annals of operations research* 134, 1 (2005), 19–67.

[6] Rina Dechter and Judea Pearl. 1985. Generalized Best-First Search Strategies and the Optimality of A*. *J. ACM* 32, 3 (July 1985), 505–536. https://doi.org/10.1145/3828.3830

[7] Nikolaus Hansen. 2016. The CMA evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772* (2016).

[8] Mark Harman and Phil McMinn. 2010. Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (March–April 2010), 226–247.

[9] T. Hohm and E. Zitzler. 2007. Modeling the Shoot Apical Meristem in A. thaliana: Parameter Estimation for Spatial Pattern Formation. In *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics (evoBIO 2007) (LNCS)*, Elena Marchiori, Jason H. Moore, and Jagath C. Rajapakse (Eds.), Vol. 4447. Springer, 102–113. https://doi.org/10.1007/978-3-540-71783-6_10

[10] Steven M. LaValle. 2006. *Planning algorithms*. Cambridge University Press. https://doi.org/10.1017/CBO9780511546877

[11] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 3–13.

[12] Maxim Likhachev, Geoffrey Gordon, and Sebastian Thrun. 2003. ARA*: Anytime A* with Provable Bounds on Sub-Optimality, Vol. 16.

[13] Jan Malburg and Gordon Fraser. 2011. Combining Search-Based and Constraint-Based Testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, USA, 436–439. https://doi.org/10.1109/ASE.2011.6100092

[14] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156. https://doi.org/10.1002/stvr.294 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.294

[15] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. https://doi.org/10.1145/2568225.2568254

[16] Man Xiao, Mohamed El-Attar, Marek Reformat, and James Miller. 2007. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering* 12, 2 (2007), 183–239.