A Case for Multi-Objective Search in Automated Patch Generation

Rahul Krishna, *Department of Computer Science*
North Carolina State University, Email: rkrish11@ncsu.edu

1 Motivation

Automated Patch generation has garnered a lot of attention in the past decade. Specifically, tools like GenProg have seen tremendous success. In fact, GenProg is now used as a baseline to compare other new techniques. Albeit popular, there is a general lack of agreement on the need for complex tools like GenProg to generate effective patches. While some researchers [1] opine that GenProg is quite effective in generating patches, given the existence of test cases, others [2]–[4] disagree. They say that it is quite unclear if the the efficacy of GenProg is merely due to its "mutate" operator or due to genetic programming.

This issue was examined by Qi et at. [2]. With their tool RSRepair (*R*andom-*S*earch *Repair*), they demonstrated the benefit of random search over GenProg.

It is my hypothesis that the lack of efficiency of GenProg in this area is not due to the failure of Genetic Algorithms (or any evolutionary computation scheme for that matter), but due to incorrect and perhaps even incomplete modeling of the search space.

As a motivating example consider Genetic Programming. The algorithm works by trying to maximize the "fitness" score. In this case, the number of passing test cases given a patch. RSRepair, on the other hand, considers either repair effectiveness (as measured by the number patch trials required) or repair efficiency (measured by the number of test case executions). It not hard to see that each of these objectives namely, the number of passing test cases, the repair effectiveness, and the repair efficiency, are all important. Perhaps some are more important than others, but they all play a crucial role nonetheless.

Therefore, I conjecture that the research in the area of automated patch generation will be best guided by modeling the domain as a multiobjective search problem with the aforementioned objectives. This, in my opinion, will allow us to leverage the wealth of research in Search Based Software Engineering (referred to henceforth as *SBSE*). To this end, this project asks the following two research questions:

RQ1: Can automated patch generation be modeled as a multiobjective search problem?

Although it makes theoretical sense to view patch generation as containing multiple objectives, this question asks if it is practical to model it as such.

RQ2: If yes, what is the best multiobjective search to generate patches automatically?

If the answer to RQ1 is affirmative. This question aims to explore various multiobjective optimizers to identify the best algorithm. Specifically, this project will compare three standard SBSE techniques: NSGAII, MOEAD, Genetic Programming, using Random Search as a baseline. In addition to this, the usefulness of a promising new algorithm called SWAY[1] [5], [6] will be also be investigated.

2 An Overview of Patch Generation

Generally, when a bug is reported, an automated program repair scheme undertakes the following three steps:

1) *Fault localization:* These techniques are used to identify suspicious faulty code snippet that may a have caused the bug.
2) *Patch Generation:* Once faulty code snippet has been located, many candidate patches can be generated through the modifications to that code snippet, according to specific repair rules based on either evolutionary computation or code-based contracts.
3) *Patch Validation:* When a candidate patch has been produced, regression testing, inclusive of negative test cases (reproducing the fault) and positive test cases (characterizing the normal behaviors), is commonly used to validate the correctness of produced candidate patch.

The above procedure can be iterated over and over again until some valid patch is found. Any patch passing all these test cases is considered valid.

3 A (very) Brief Overview of Multiobjective Search

A search is the task of finding one or more solutions which satisfy one or more specified objectives. While a single-objective optimization involves a single objective function and a single solutions, a multiobjective optimization considers several objectives simultaneously. In reference to the current problem

1. This algorithm is in current development by our research group (see http://ai4se.net).

Measure	Description
Runtime	Time taken for the algorithm to be generate optimal solutions.
Speed Up	Ratio of time taken for the parallelized version of the algorithm over the time taken for the serial version.
Convergence	Convergence represents the accuracy of the obtained solutions. It is the distance between the obtained solutions and ideal Pareto frontier.
Diversity	Diversity represents the spread of the proposed solutions. Ideally the solutions should be well distributed across the Pareto frontier,rather than concentrated in certain regions.

Figure 1. Performance Measures

we have three competing objectives, all of which need to be maximized: (1) Number of passing test cases; (2) The repair effectiveness; and (3) The repair efficiency

In such a case, a multiobjective optimizer generates a set of alternate solution with certain trade-offs. These are called *Pareto Optimal Solutions*. The solutions, in our case, will be a set of valid patches that perform equally well when measured in terms of the three objectives.

Multiobjective problems are usually complex, NP-Hard, and resource intensive. Although exact methods can be used, they consume prohibitively large amounts of time and memory. An alternative approach would be to make use of metaheuristic algorithms from RQ2. These approximate the Pareto frontier in a reasonable amount of time.

4 Evaluation

4.1 Metrics

Figure 1 highlights some of the preliminary performance evaluation measures we use to evaluate the algorithms. Please note that this will be expanded in future to accommodate other measures.

4.1.1 How to calculate them?

Calculating these metrics are well studied [7], [8]. We have an implementation in place that can readily give these metrics, given a set of ideal solutions (in this case, valid patches).

4.2 Datasets

The data set required for evaluating will be obtained from some recent work on GenProg [1]. My reading of the literature suggests that these dataset are the "benchmarks", used by several researchers [2]. These are C programs with all the bugs reported reported in the historical versions as tabulated in Figure 2.

4.3 Tools

As for tools, it is worth noting that AST can be generated from C programs using a tool called CIL[2].

2. https://sourceforge.net/projects/cil/

Program	LOC	Test Cases	Version
`lighttpd`	62,000	5	bug-1806-1807
		16	bug-1913-1914
		18	bug-2330-2331
		17	bug-2661-2662
`libtiff`	77,000	73	bug-01209c9-aaf9eb3
		31	bug-0860361d-1ba75257
		73	bug-0fb6cf7-b4158fa
		33	bug-10a4985-5362170
		59	bug-4a24508-cc79c2b
		64	bug-5b02179-3dfb33b
		59	bug-6f9f4d7-73757f3
		77	bug-829d8c4-036d7bb
		35	bug-8f6338a-4c5a9ec
		76	bug-90d136e4-4c66680f
		73	bug-d39db2b-4cd598c
		76	bug-ee2ce5b7-b5691a5a
		60	bug-0661f81-ac6a583
		31	bug-3af26048-72391804
		33	bug-d13be72c-ccadf48a
`gmp`	145,000	144	bug-14166-14167
`python`	407,000	303	bug-69783-69784
`gzip`	491,000	2	bug-3fe0caeada6aa3-39a362ae9d9b00
`php`	1,046,000	4,986	bug-309892-309910
`wireshark`	2,814,000	53	bug-37112-37111

Figure 2. Subject Programs. Image courtesy [2].

4.4 Analysis Plan

To provide a more comprehensive comparison, the optimizers will be ranked ranked statistically. To do this, I shall use the Scott-Knott procedure recommended by Mittas & Angelis [9].

This method sorts a list of l treatments with ls measurements by their median score. It then splits l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

$$E(\Delta) = \frac{ms}{ls} abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} abs(n.\mu - l.\mu)^2$$

Scott-Knott then applies some statistical hypothesis test H to check if m, n are significantly different. If so, Scott-Knott then recurses on each division.

5 Proposed Timeline

Figure 3 highlights the proposed timeline for the tasks undertaken in the project. Please note again that this subject to change, please see https://github.com/ai-se/spatch/milestones for the latest milestone progress.

Milestone	Task	Timeline
1	Setup problem domain: Get and verify datasets	10/18/16
2	Create a python package that implements the five algorithms	10/28/16
3	Establish fitness measurements on the dataset. This maybe a tedious task and my take the longest time.	11/18/16
4	Generate Pareto Optimal solutions and run validations	11/21/16
5	Assess results in terms of the Research Questions and prepare a report.	11/25/16

Figure 3. Timeline

References

[1] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 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)*, pages 3–13. IEEE, 2012.

[2] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[3] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[4] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10. IEEE, 2011.

[5] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. Is" sampling" better than" evolution" for search-based software engineering? *arXiv preprint arXiv:1608.07617*, 2016.

[6] Vivek Nair, Tim Menzies, and Jianfeng Chen. An (accidental) exploration of alternatives to evolutionary algorithms for sbse. In *International Symposium on Search Based Software Engineering*, pages 96–111. Springer, 2016.

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[8] Tushar Goel and Nielen Stander. A study on the convergence of multiobjective evolutionary algorithms. In *Preprint submitted to the 13th AIAA/ISSMO conference on Multidisciplinary Analysis Optimization*, pages 1–18, 2010.

[9] Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Trans. Software Eng.*, 39(4):537–551, 2013.