The Impact of Search Algorithms in Automated Program Repair

Fatmah Yousef Assiri*, James M. Bieman

Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, U.S.A.

Abstract
Automated program repair (APR) techniques locate and fix software faults automatically. In order to fix faults, APR applies a set of program modification operators (PMOs) to modify faulty programs. A potential repair is found when APR applies a PMO that fixes a fault. A brute-force search algorithm applies all PMOs in a predefined order until a potential repair is found. Brute-force can guarantee a fix but lowers APR performance, especially when it uses many PMOs. Stochastic search algorithms, such as a genetic algorithm, efficiently search the modifications space for a PMO that fixes a fault. In this paper, we conduct a comprehensive evaluation of the impact on APR effectiveness, APR performance, and the quality of potential repairs of three stochastic search algorithms: (1) a genetic algorithm (GA), (2) a genetic algorithm without a crossover operator (GAWoCross), and (3) a random search (RS). Our evaluation using 41 faulty versions of six different C programs shows that RS improves APR effectiveness and performance, but GA and GAWoCross improve the quality of potential repairs by generating more validated repairs, and potential repairs that failed fewer regression tests compared to RS.

© 2015 The Authors. Published by Elsevier B.V.

Keywords: automated program repair; search algorithms; genetic algorithm; random search; effectiveness; performance; repair quality

1. Introduction
Automated program repair (APR) techniques locate and fix software faults without human interference. These techniques take a faulty program, a set of repair tests, and a list of potentially faulty statements (LPFS) that are created by applying a fault localization technique in a pre-processed step. To modify faulty programs, APR uses a set of program modification operators (PMOs) and applies them to faulty statements generating new versions of the faulty program called variants. APR applies a search algorithm to select a PMO; some search algorithms run for multiple iterations. Each variant is executed against a set of repair tests. If a variant passes all repair tests, it represents a potential repair. Potential repairs that pass other tests that were not included in repair tests are called validated repairs. Figure 1 describes the overall process for APR.

* Corresponding author. Tel.: +1-970-960-2718
E-mail address: fatmahya@cs.colostate.edu

1877-0509 © 2015 The Authors. Published by Elsevier B.V.
APR components and mechanisms have a major impact on APR effectiveness, performance and repair quality. APR effectiveness is the ability to find potential repairs and the frequency of successful potential repair. APR performance is an external attribute that is measured by the time and steps required to find potential repairs. Repair quality is studied in terms of repair correctness; repair correctness is concerned with how well the potential repaired program retains required functionality without the introduction of new faults or missing actual faults.

Search algorithms are used to select a PMO to fix a fault (Step 2 in Figure 1). There are two search algorithm categories: exhaustive and stochastic searches. APR is most effective with an exhaustive brute-force search since it is guaranteed to repair faults related to one of the PMOs, but at a cost of lowered performance. An exhaustive search algorithm can be infeasible with large programs especially when coupled with many PMOs. On the other hand, an APR using a stochastic search algorithm, such as a genetic algorithm, can more efficiently search the modification space for a PMO, but it might never introduce a PMO that fixes a fault.

Both genetic algorithm and brute-force algorithms have been used for APR. Le Gouse et al. developed the GenProg tool, which uses a genetic algorithm to fix faults. They evaluated the impact of the use of a crossover operator and found that removing the crossover operator decreased the time required to fix faults but lowered the effectiveness. Qi et al. found that random search improved the effectiveness and performance of GenProg compared to a genetic algorithm (GA). Both studies used GenProg, which applies only three PMOs. Debroy and Wong applied brute-force and many PMOs to repair faults. To manage the performance, they proposed limiting the number of PMOs, the number of PMOs applied to each statement, or the number of potentially faulty statements.

In this paper, we study the use of stochastic search algorithms to improve APR performance when many PMOs are used. We evaluate the impact of different search algorithms on APR quality factors (effectiveness, performance, and repair quality) using our MUT-APR repair tool, which was built by adapting GenProg. MUT-APR applies a different set of PMOs than those used by GenProg. MUT-APR PMOs, adapted from Debroy and Wong, fix faults by transforming faulty operators into alternatives. It applies PMOs that fix faults in relational operators, arithmetic operators, shift operators and bitwise operators in many different program constructs including if, return, assignment, and loops.

We apply three stochastic search algorithms: (1) genetic algorithm (GA), (2) genetic algorithm without a crossover operator (GAWoCross), and (3) random search (RS). A genetic algorithm applies a set of PMOs to modify faulty program by adding a change to a faulty statement creating variants. A calculated fitness value for each variant determines the goodness of the variant. Then, a selection algorithm selects variants with the best fitness values for use in the next generation. A crossover operator combines two variants to generate two new child variants. However, since MUT-APR applies simple PMOs to modify a faulty program, there should be no advantages from applying a crossover operator. To test this thesis, we evaluate APR quality factors applying a genetic algorithm without a crossover operator. Then, to study the influence of randomness in fault fixing, we apply a basic random algorithm.
that does not apply either a selection algorithm or a crossover operator, we simply refer to this algorithm as random search. All three algorithms select PMOs randomly from the pool of PMOs. MUT-APR assigns equal probabilities to all PMOs, thus the probability of selecting a particular PMO is \( p = 1 / \text{Total Number of PMOs} \).

We evaluate our work using the Siemens Suites programs and two larger programs, space and sed, from the Software Infrastructure Repository (SIR). APR effectiveness is measured by the success rate; success rate is the percentage of trials that result in potential repairs. One trial is equivalent to one execution of the repair algorithm until a potential repair is found or a predefined parameter limit is reached. APR performance is measured by the average number of invalid variants (ANGV), and the average total time (Time) until a potential repair is found. Repair correctness is measured by the percentage of failed potential repairs (PFR), and the average percent of failed regression tests for \( N \) potential repairs for each faulty version (APFT); \( N \) is the number of potential repairs for 100 trials. To the best of our knowledge, we are the first to study the impact of different search algorithms on the quality of generated repairs. Our results show that the RS algorithm improves success rates and produces fewer variants compared to both versions of genetic algorithm, but genetic algorithms produce more validated repairs and removing the crossover operator (GAWoCross) generate potential repairs that fail fewer regression tests compared to alternatives.

The main contributions of this paper are the following:

- Search algorithm and APR effectiveness: RS improved APR success rate compared to both GA and GAWoCross.
- Search algorithm and APR performance: RS improved APR performance by requiring fewer variants to find a potential repair compared to the other genetic algorithms.
- Search algorithms and repair quality: Genetic algorithms produced higher quality repairs compared to RS.

2. Search Algorithms

Program modification operators can be selected in a predefined order as done by a brute-force search method, or randomly as done by stochastic search. Brute-force involves an exhaustive search which is inefficient, and can be infeasible with large programs. We use stochastic search algorithms to more efficiently search for a PMO that fixes a fault.

Genetic Algorithm (GA)

Genetic programming is an evolutionary search method that evolves computer software by applying a genetic algorithm (GA). A genetic algorithm addresses the combinatorial optimization problems. Genetic programming has been used to automate fault fixing.

Algorithm derived from the one used by GenProg, describes how the genetic algorithm in MUT-APR can fix faults. MUT-APR takes a faulty C program, a set of repair tests, a list of potentially faulty statements (LPFS), and a set of parameters: population size (\( \text{pop}_\text{size} \)), number of generations (\( \text{gen} \)), and maximum fitness value (\( \text{max} \)). The initial population, which consists of program variants, of size \( \text{pop}_\text{size} \), is created by mutating a mutable faulty statement (line 3). A mutable faulty statement is a program statement in the LPFS that contains one or more operators that can be mutated by MUT-APR. PMOs are selected randomly from the pool of PMOs, and statements are selected sequentially based on the order given by the LPFS. If a statement is not mutable, or the selected PMO is not one of the alternatives of the faulty operator, the original faulty program is retained. Otherwise, a new version of the faulty program is created. The fitness function is computed for each variant (line 4) to determine if the generated variant is a repair or not.

The fitness function is given in Equation \( \text{fitness} = |\text{Tests}_{\text{pass}}| \cdot \text{W}_{\text{pass}} + |\text{Tests}_{\text{fail}}| \cdot \text{W}_{\text{fail}} \). Tests_{\text{pass}} and Tests_{\text{fail}} are the number of passing and failing tests respectively, and \( \text{W}_{\text{pass}} \) and \( \text{W}_{\text{fail}} \) are positive constants that represent the weights of passing and failing tests respectively. Failing tests are assigned a weight of 10 and passing tests are assigned a weight of 1. A variant (v1) that passes failing tests will have a higher fitness value than another variant (v2) that passes all passing tests but not failing tests. Thus, v1 will have a higher chance to be used for the next generation than v2. If a variant that maximizes the fitness function (equal to \( \text{max} \), which is one input to the algorithm) is found, a repair is found and the process stops (line 20).
Algorithm 1 Genetic Algorithm Pseudocode

1: Inputs: Program \( P \), max, gen, pop_size, LPFS
2: Output: variant
3: let \( pop = initial\_pop(P, pop\_size) \)
4: let \( fitness = ComputeFitness(pop) \)
5: let \( len = List.length(LPFS) \)
6: repeat
7: let \( variants = select(pop) \)
8: if \( size(variants) < pop\_size/2 \) then
9:   let \( variants = double(variants) \)
10: end if
11: for all two variants \( (v_{p1}, v_{p2}) \) \( \in \) variants do
12:   let \( new\_Variants(v_{c1}, v_{c2}) = crossover(v_{p1}, v_{p2}) \)
13:   let \( new\_Pop = v_{c1}, v_{c2}, v_{p1}, v_{p2} \)
14: end for
15: let \( i = 0 \)
16: for all variant in \( new\_Pop \) do
17:   let \( stmtId = LPFS[i] \)
18:   let \( pmo = choose(PMO) \)
19:   let \( pop = apply(variant, stmtId, pmo) \)
20:   let \( fitness = ComputeFitness(pop) \)
21: if \( i != len-1 \) then
22:   \( i++ \)
23: end if
24: end for
25: until \( fitness = \max \| size(Pop) = pop\_size \)
26: return variant

If no potential repair is found, variants with a fitness equal to zero or that do not compile are discarded, and the algorithm selects variants with higher fitness values to continue the process (line 7). GA requires \( pop\_size/2 \) variants to continue the evolution process. Thus, if the number of remaining variants is less than half the population size \( pop\_size/2 \) (line 8), variants are duplicated (line 9) so that the number of variants is equal to \( pop\_size/2 \) for use by the crossover operator (line 11-14).

The crossover operator (line 12) combines information from two variants to create two children. We applied a one-point crossover, which selects a random cut-off point, and swaps the parents’ statements after the selected point to create children variants. All parents \( (v_{p1}, v_{p2}) \) and children \( (v_{c1}, v_{c2}) \) are included in the new population (line 13). Then, a PMO is applied to each variant (line 18) to create the population for the next generation (line 19). Fitness is computed for each member of the population (line 20). If a variant that maximizes the fitness functions is found (line 25), the process stops and the variant is returned (line 26). The algorithm runs for multiple iterations. Each iteration consists of one GA loop. The genetic algorithm stops when a potential repair is found, or the number of iterations exceeds the upper bound. We set an upper bound on the number of iterations that is equal to \( gen \) (\( gen \) is one input to the repair algorithm).

Genetic Algorithm Without a Crossover Operator (GAWoCross)

A genetic algorithm involves a selection algorithm, program modification operators, and a crossover operator. To study the influence of the crossover operator in fault fixing, we remove the crossover operator from the genetic algorithm. The new algorithm is called GAWoCross.

GAWoCross is similar to the genetic algorithm in Section 2 except that (1) we do not implement a crossover operator (lines 11-14 from algorithm 1), and (2) the number of variants required to continue the process must be equal to the number of \( pop\_size \) not \( pop\_size/2 \) as done in GA. Variants of the initial population are created, and a fitness
function is computed as before. Then, a selection algorithm is applied to select the variants with better fitness values for use in the next generation. If the number of remaining variants (line 8 in algorithm 1) is less than the population size (\( \text{pop.size} \)), the algorithm creates more copies of the generated variants so that the number of variants is equal to the \( \text{pop.size} \). Then, PMOs are applied to each variant (lines 16-23 in algorithm 1). The algorithm is repeated for many generations until the number of generations reaches its limit or a potential repair is found.

**Random Search (RS)**

Our random search does not apply a selection algorithm or crossover operator as done by both GA and GAWoCross. It applies PMOs randomly until a repair is found or the population size reaches a set limit. RS will generate variants by selecting statements sequentially from the LPFS, then apply a PMO. Our RS algorithm differs from the RS algorithm implemented by Qi et al. (12): (1) our RS search runs for one generation to ensure each variant contains one change, while their RS search runs for multiple iterations, and (2) our algorithm computes a fitness function to validate variants, while their algorithm runs variants on repair tests and discards variants as soon as one test fails.

**Algorithm 2 Random Search Pseudocode**

```plaintext
1: Inputs: Program \( P \), max, \( \text{pop.size} \), LPFS
2: Output: variant
3: let \( i = 0 \)
4: let \( \text{len} = \text{List.length}(\text{LPFS}) \)
5: repeat
6: let \( \text{stmtId} = \text{LPFS}[\text{i}] \)
7: let \( \text{pmo} = \text{choose}(\text{PMO}) \)
8: let \( \text{Pop} = \text{apply}(\text{P, stmtId, pmo}) \)
9: let \( \text{fitness} = \text{ComputeFitness}(\text{pop}) \)
10: \( i++ \)
11: if \( i = \text{len} - 1 \) then
12: let \( i = 0 \)
13: end if
14: until \( \text{fitness} = \text{max} \| \text{size(Pop)} = \text{pop.size} \)
15: return variant
```

Algorithm 2 describes how our random algorithm RS in MUT-APR can fix faults. RS selects a statement from the LPFS (line 6). Then, it selects a PMO randomly (line 7) to create a new variant (line 8). The variant is created by mutating faulty statements that are mutable. Otherwise, the original program is retained.

Fitness is computed for each variant by checking if the variant passes the repair tests or not (line 9). The fitness function computes the number of passing and failing tests for each variant (Equation 2). The fitness function for the RS is different than the one used by GA and GAWoCross. RS does not apply a selection algorithm, thus the algorithm does not favor a variant that passes failing tests over another variant that passes passing tests as done the GA. Thus, there is no need to assign different weights for passing and failing tests. The process continues until a variant that passes all repair tests is found, or the number of generated variants is equal to \( \text{pop.size} \) (line 14).

\[
\text{fitness} = |\text{Tests}_{\text{pass}}| + |\text{Tests}_{\text{fail}}|
\]

(2)

3. Evaluation

We designed experiments to answer the following research questions:

- **RQ1**: What is the relative APR effectiveness when different search algorithms are employed?
- **RQ2**: What is the relative APR performance when different search algorithms are employed?
- **RQ3**: Does the use of different search algorithms affect the quality of potential repairs?
Table 1. Benchmark programs. Each Program is an original program from the SIR\textsuperscript{5}. LOC is the number of lines of codes. #Faulty Ver. is the number of faulty versions. Rep. Tests is the average number of repair tests for each faulty version. Regr. Tests is the number of regression tests for each faulty version. LPFS is the average number of statements in the list of potentially faulty statements (LPFS).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>173</td>
<td>13</td>
<td>5.8</td>
<td>1608</td>
<td>60</td>
</tr>
<tr>
<td>replace</td>
<td>564</td>
<td>10</td>
<td>17.2</td>
<td>5542</td>
<td>178</td>
</tr>
<tr>
<td>schedule2</td>
<td>374</td>
<td>6</td>
<td>7</td>
<td>2710</td>
<td>97.3</td>
</tr>
<tr>
<td>tot_info</td>
<td>565</td>
<td>9</td>
<td>8</td>
<td>1052</td>
<td>92.9</td>
</tr>
<tr>
<td>space</td>
<td>6195</td>
<td>2</td>
<td>26</td>
<td>5670</td>
<td>619</td>
</tr>
<tr>
<td>sed</td>
<td>14427</td>
<td>1</td>
<td>38</td>
<td>370</td>
<td>1388</td>
</tr>
<tr>
<td>Total</td>
<td>22201</td>
<td>41</td>
<td>102.0</td>
<td>16952</td>
<td>3921.2</td>
</tr>
</tbody>
</table>

3.1. Evaluation Design

Our evaluation applies different search algorithms to repair faulty operators on Siemens Suites\textsuperscript{5} and two other large programs: space and sed. We used many faulty versions of each subject program selected from the Software-Artifacts Infrastructure Repository (SIR\textsuperscript{5}). We also used the C mutation tool Proteum/IM 2.0\textsuperscript{13} to create additional faulty versions. Each faulty version is seeded with a single operator fault. For each faulty version, we selected a set of repair tests that satisfy the branch coverage criterion since our previous study\textsuperscript{14} showed that the use of branch coverage repair tests produces more validated repairs. We used the Ochiai fault localization technique to generate the LPFS since our evaluation of the impact of different fault localization techniques on APR showed that Ochiai is one of the best techniques to place faulty statements near the head of the LPFS. Table 1 includes a list of the subject programs along with their sizes in LOC, the number of faulty versions, the average number of repair tests, the number of regression tests, and the average number of statements in the LPFS.

Our evaluation used our repair tool MUT-APR implementation of three algorithms: (1) genetic algorithm (GA), (2) genetic algorithm without a crossover operator (GAWoCross), and (3) random search (RS). Since the algorithms select PMOs randomly, we ran each algorithm 100 times on each faulty version so that it is more likely to fix a fault in at least one execution; each execution is called a trial. Then we compared APR effectiveness, APR performance, and repair quality by computing the average of the 100 trials. Each trial of GA and GAWoCross consisted of many generations/iterations of the genetic loop. Each generation consisted of a population size that is equal to the number of potentially faulty statements in the LPFS. We investigated the use of five and ten generations with the genetic algorithm (GA). Increasing the number of generations improved the success rate. The average success rate improvement is 21.8%. Since we generated a large population for each generation, we executed the genetic algorithms for only five generations and compared the results. The random search (RS) runs for one generation since it does not apply a selection or crossover operator. Thus, each execution of RS consisted of one large generation and population size of $5 \times |\text{LPFS}|$.

3.2. Evaluation Results

We analyzed our results to answer the three research questions. To assess effectiveness, we computed the success rate. The success rate is the number of trials resulting in a potential repair. An algorithm that generates more repairs improves the success rate. To assess APR performance, we computed the average number of invalid variants (ANGV) and the average total time until a potential repair is found (lower is better). To assess repair quality we measure the percent of failing potential repairs (PFR) and the average percent of failing regression tests (APFT). PFR is the percent of potential repairs that failed at least one regression test, and APFT is the average percent of failed regression tests for $N$ potential repairs for each faulty version; $N$ is the number of potential repairs for 100 trials (lower is better). Due to the space limits, we could not include the result distributions.
Table 2. Results summary when different search algorithms were applied by MUT-APR. Succ is the success rate over 100 trials. ANGV is the average number of invalid variants, and Time is the average total time until a potential repair is found. PFR and APFT are the percent of failing potential repairs and the average percent of failing regression tests for N potential repairs for each faulty version.

<table>
<thead>
<tr>
<th></th>
<th>Succ</th>
<th>ANGV</th>
<th>Time</th>
<th>PFR</th>
<th>APFT</th>
<th></th>
<th>Succ</th>
<th>ANGV</th>
<th>Time</th>
<th>PFR</th>
<th>APFT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>10.3</td>
<td>491.5</td>
<td>36.60</td>
<td>35.0</td>
<td>26.6</td>
<td>GAWoCross</td>
<td>14.8</td>
<td>588.7</td>
<td>51.60</td>
<td>39.2</td>
<td>5.4</td>
<td>RS</td>
</tr>
<tr>
<td>Median</td>
<td>7.00</td>
<td>319.0</td>
<td>6.700</td>
<td>9.00</td>
<td>12.4</td>
<td></td>
<td>13.0</td>
<td>366.4</td>
<td>4.300</td>
<td>22.2</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Std dev.</td>
<td>10.3</td>
<td>697.8</td>
<td>133.4</td>
<td>42.7</td>
<td>25.7</td>
<td></td>
<td>9.5</td>
<td>581.2</td>
<td>200.3</td>
<td>43.5</td>
<td>9.1</td>
<td></td>
</tr>
</tbody>
</table>

3.2.1. APR Effectiveness

To evaluate APR effectiveness when different search algorithms were used, we compared the average success rate for each algorithm across all faulty versions (Table 2). The average success rate for GA is 10.3%, for GAWoCross is 14.8%, and for RS is 15.2%. RS algorithm improved the average success rate compared to both versions of genetic algorithm (GA and GAWoCross). Thus, randomness of stochastic search algorithms increased the average success rate; however, the difference between RS and GAWoCross average success rate is only 0.4%. These results differ from those found using GenProg [12]; they found that removing the crossover operator decreases success rate.

Then we compared the average success rate for each subject program. We found that, for tcas and schedule2 (19 out of the 41 faulty versions), GAWoCross improved the average success rate compared to RS and GA, and, for faulty versions of space and sed, GA produced a higher success rate. We applied the Wilcoxon Signed-Rand Test to analyze the improvement of success rates when different search algorithm were used. The difference is statistically significant between RS and GA (p-value = 0.001), but the difference between RS and GAWoCross is not significant (p-value= 0.452) at the 0.95 confidence level.

3.2.2. APR Performance

To assess APR Performance, we compared the ANGV and the time required to fix faults (Table 2). GA generated an average of 491.5 variants to find a potential repair, and GAWoCross and RS generated an average of 588.7 and 378.7 variants, respectively. For 50 faulty versions (all except sed), RS generated fewer variants compared to GA and GAWoCross. We also compared search algorithms using the time metric. The average time required to fix faults were 36.6, 51.6 and 36.6 seconds for GA, GAWoCross and RS, respectively. RS and GA required less time to fix faults compared to GAWoCross. We expected that GA and RS would take less time compared to the GAWoCross since they generated fewer variants in almost all trials. We analyzed the results by applying Wilcoxon Signed-Rand Test. The performance differences between the three algorithms are not significant at the 0.95 confidence level, but they provide a strong evidence that RS and GA improved MUT-APR performance.

3.2.3. Repair Quality

We compared repair correctness by measuring the average PFR and the APFT. PFR indicates the relative number of failed potential repairs. A failed potential repair is one that fails at least one regression test. The average PFR for GA is 35%. The average PFR for GAWoCross and RS is 39.2% and 36.6% (Table 2). Then, we compared the average PFR for each subject program. For 25 out of the 41 faulty versions, GA produced fewer failing potential repairs, and for tot_info faulty versions GAWoCross produced fewer failing potential repairs. Thus, the use of the selection algorithm and crossover operator in GA guide the search algorithm toward the validated repair, which produced a greater number of validated repairs. The difference between GA and GAWoCross is not significant (p-value = 0.363), but the difference between GA and RS is significant (p-value=0.001) at the 0.95 confidence level.

For each failing potential repair, we measured the average number of failing regression tests (APFT). This measure estimates how far a failing potential repair is from being a validated repair. The APFT average for each search algorithm is 26.6, 5.4, and 18.6 for GA, GAWoCross and RS, respectively. Thus, GAWoCross produced potential repairs that failed fewer regression tests compared to the GA and RS algorithms. We applied Wilcoxon Signed-Rand Test to compare the difference between means of PFR and APFT at 0.95 conference level; the difference is significant between GAWoCross and both GA (p-value=0.0007) and RS (p-value=0.005).
4. Threats to Validity

Our study evaluated the impact of different search algorithms on the performance, effectiveness, and repair quality of APR. Our study was conducted using the Siemens Suites and two larger programs. To mitigate threats to internal validity, we selected many faulty versions of each subject program; each faulty version were seeded with one simple mutation fault. External validity relates to the ability to generalize the results. Our evaluation consists of programs of different sizes including two large C programs (more than 14K LOC); however, our results cannot be generalize to other faults such as multiple faults or other domains. Construct validity relates to the repair test suites used to produce repairs. To mitigate this threat we used repair test suites that satisfy branch coverage, which produced higher quality repair.\[14\]. The accuracy of PFR and APFT depends on the quality of the regression tests. We used one set of regression tests from the SIR repository. Different regression tests might produce different results. Conclusion validity is another threat to our results. To mitigate the threats to the conclusion validity, we used the same subject programs and repair test suites with all search methods. We applied Wilcoxon Signed-Rand Test to study the statistical relation between variables, we also ensured randomness in the experimental setting.

5. Conclusion

Our study evaluated the impact of the different search algorithms on MUT-APR: (1) genetic algorithm (GA), (2) genetic algorithm without a crossover operator (GAWoCross), and (3) random search (RS). We found that RS had the best success rate; GA and GAWoCross improved the quality of potential repairs. To fix faults with the three algorithms, we used a population size equal to the number of statements in the LPFS ($|LPFS|$). To improve MUT-APR performance, we suggest decreasing the population size to $|LPFS|/2$, since faulty statements were ranked in the top half of the LPFS in 78% of faulty versions. We plan to implement additional search algorithms that guide the search to the correct program modification operator (PMO) by checking the faulty operator and only calling a PMO from a group of PMOs that contain the alternatives of the faulty operator, which will increase the probability of selecting a PMO that will fix a fault, and thus can improving APR process.

References