Evolving Patches for Software Repair

Thomas Ackling
University of Adelaide
Adelaide, South Australia
Australia, 5005
thomas.ackling@gmail.com

Brad Alexander
University of Adelaide
Adelaide, South Australia
Australia, 5005
brad@cs.adelaide.edu.au

Ian Grunert
Atlassian Pty Ltd
173-185 Sussex St, Sydney
NSW, Australia, 2000
igrunert@atlassian.com

ABSTRACT

Defects are a major concern in software systems. Unsurprisingly, there are many tools and techniques to facilitate the removal of defects through their detection and localisation. However, there are few tools that attempt to *repair* defects. To date, evolutionary tools for software repair have evolved changes directly in the program code being repaired. In this work we describe an implementation: pyEDB, that encodes changes as a series of code modifications or patches. These modifications are evolved as individuals. We show pyEDB to be effective in repairing some small errors, including variable naming errors in Python programs. We also demonstrate that evolving patches rather than whole programs simplifies the removal of spurious errors.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program Modification*

Keywords

Debugging,Fault-Repair,Genetic-Programming,Python

1. INTRODUCTION

Software defects are a ubiquitous concern in software systems. Once a defect is introduced into the code it has enormous potential to distress customers and vex developers. It is estimated that just the software bugs that make it through testing cost $20 billion per year in the United States alone[21].

Unsurprisingly, given this cost, a rich body of knowledge has developed to assist in the process of software defect removal. Specifically, there are many works describing tools that help *detection* of software defects. These include automatic test generators[6, 7], algorithms that assess the quality of tests[26]. Likewise there are a number of algorithms to help *locate* software defects[24, 12, 3]. There are still other systems assist with both detection and localisation, statically [5, 4, 14] and dynamically [15, 18, 8]

By contrast, there is relatively little work on the automated *repair* of code defects[2, 1, 11, 22, 23, 19, 16, 20, 25]. This is in spite of the often long lead times for manually fixing defects even when they are located[16]. Current work on automatic repair falls into three broad categories. First there are implementations to locate and repair specific types of error with distinctive signatures such as buffer overflows or heap allocation errors[19, 16]. Second there are implementations that locate and repair errors by brute-force search[20, 25]. Third there are implementations using heuristic search to locate and repair errors[2, 1, 11, 22, 23]. The implementations in this third category, have used genetic programming[13] (GP) to search the space of code-corrections, with the aim of minimising the number of test-cases failed. These implementations are applicable to a variety of errors. Interestingly, these GP solutions for software repair have all used individual variations on the original defective program as the genotype. With this encoding, care needs to be taken to limit the number of changes inserted into the original code to prevent the individuals from drifting too far from the original code.

Like the work above, we also describe a GP approach to automatic defect repair. However, in contrast to the work above, we evolve *patches* to the defective program as individuals rather than evolving variations on the programs as a whole. This encoding keeps individuals short and makes it trivial put an upper bound on the number of corrections expressed thus strictly limiting the number of spurious corrections that can arise.

The implementation described here is extensible to any repair expressible as a rewrite-rule. This makes our framework, at least potentially, very general. However here, in the first instance, we demonstrate our approach for two different types of point repairs: replacing incorrect relational operators ($<, \leq, >, \geq, ==$); and replacing incorrect variable names. We choose these two types of repair because they are simple, relatively common and require contrasting approaches to generating rewrite rules[1].

Our implementation, which we call *pyEDB* (python evolutionary debugger), is written in Python and corrects defects in Python applications but its underlying algorithm is non-language specific. Python was chosen due to ease of implementation – most of the modules we require are

[1]Rewrite rules for relational operators can be generated statically, rewrite rules for variable name errors must by generated dynamically.

part of the standard library, including Abstract-Syntax-Tree (AST) compilation and modification, and tracing of execution paths of programs.

The contributions of this work are as follows. We express defect fixes as short program patches rather than changes embedded in a program (section 4.1); this expression of changes as patches allows genetic operators to work on individual changes rather than programs as a whole. We describe a genotype-to-phenotype mapping (section 4.2) that reduces the search-space by avoiding changes giving rise to syntax and variable naming errors. We demonstrate that by using patches, controlling the number of changes is trivial(section 4.4). We describe a simple and efficient algorithm for removing any spurious changes from the patch that is ultimately selected (section 5.3) thus minimising the extent of changes needed for repair.

The remainder of this article is structured as follows. Next we review related work. In section 3 we precisely state the problem we are solving. In section 4 we describe the algorithm used by PyEDB. In section 5 describe our experimental setup and present our results and, finally, in section 6 we conclude and canvass future work.

2. RELATED WORK

The most closely related work in software repair is that of Arcuri[2, 1] and Weimer and Forrest[11, 22]. In both cases the authors use GP to evolve variants on the original defective program, both utilised fault localisation techniques to help locate errors, and in both frameworks extra care had to be taken to ensure that individuals did not diverge too much from the original program. Arcuri's JAFF framework introduced diverse mutations into the original defective program. To prevent the evolved programs from diverging too much from the original code Arcuri first took the approach of re-introducing the original program into the population at each generation[1]. Later he simply limited the number of nodes that could be added or modified in any single mutation operation[2]. Arcuri also introduced a novel way to co-evolve tests and code in a predator/prey relationship.

Arcuri's framework can make general changes to small programs. In contrast Weimer and Forrest dealt with much larger programs but limited changes to deleting, moving and copying extant lines of code Weimer and Forrest took a different approach. Dealing with much larger programs, their framework[11, 22] limited changes to deleting, moving and copying existing lines of code. Divergence was limited by "crossing-back" individuals with the original program. Weimer and Forrest also utilised delta-debugging[24] as a post-processing step to help cull unnecessary changes from the fittest individual. The cost of this post-processing rises with the size of the program whereas in pyEDB, the maximum amount of post processing is a constant bounded by the short patch genome.

Wilkerson[23] used GP to co-evolve C++ applications in competition with sets of test cases. As with previous approaches, individuals were expressed as variants of the original defective program.

Outside the field of evolutionary search Sidiroglou[19] describes a system that adaptively patches servers against zero-day exploits by worms. Novark[16] presents a tool to patch memory allocation errors. These systems have the common feature of addressing specific issues with a recognisable signature.

Finally, there is a literature of brute-force approaches to program debugging ([20, 25] are two examples). Like ours, these approaches are able to strictly limit the number and type of changes but the brute-force search limits their applicability to smaller programs.

3. PROBLEM STATEMENT

Given an abstract syntax tree for a program P and a list of tests $ts = [t_0, \ldots, t_{n-1}]$ derive a list of modifications (a patch) $mds = [md_0, \ldots, md_{m-1}]$ that, when applied to P minimises the number of tests failed. More formally we want to minimise:

$$fc = \mathit{testsFailed}(\mathit{applyMods}(rs, P), ts)$$

where $\mathit{testsFailed}(P', ts)$ returns a non-negative integer representing the number of tests P' fails when applied to ts and $\mathit{applyMods}(mds, P)$ applies the repairs in mds in turn to P producing a new, possibly better, program P'.

Ideally, we expect our pyEDB framework, for a given benchmark P, to be *effective* in evolving a list of modifications; *efficient* in evolving modifications in reasonable time; *accurate*, that is, not induce spurious changes and not overfit the test data.

4. THE ALGORITHM

This section describes how patches are evolved in pyEDB. It is divided into the following parts: encoding the modifications (next); genotype to phenotype mapping (section 4.2); biasing the mapping to speed up search (section 4.3); and a description of evolutionary framework (section 4.4).

4.1 Encoding of Modifications

Each modification md_i in the modification list mds takes the form of a pair:

$$md_i = (loc_i, targ_i)$$

the first element of this pair loc_i is the unique location of an AST node in P and the second element $targ_i$ is the new value for the node after the modification is applied. Each modification md_i in mds is applied in sequence. In our current configuration, if the are two modifications md_i and md_j in mds that modify the *same* location, the second of these to appear in the list overwrites the first one[2]. As an example of how mds can be applied consider the following (incorrect) python code snippet for bounding access into an array `a`:

```
if x > length:
  return -1
else:
  return a[a]
```

maps to the AST in Figure 1(a). This tree contains two small errors. A wrong operator `>` at node 1 and an incorrect index-value `a` at node 9. A modification list that will fix this tree is $mds = [(1, >=), (9, x)]$ produces the tree in Figure 1(b). This tree corresponds to the correct code:

```
if x >= length:
  return -1
else:
  return a[x]
```

[2]Addresses are are interpreted relative to the AST in its original form before any modifications are applied – changes to any previously modified part of the tree are ignored.

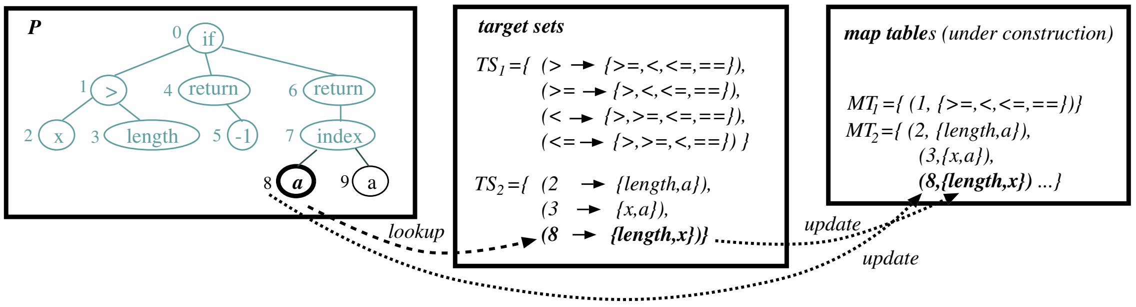

Figure 2: Snapshot of the construction of the mod-tables for the AST from Figure 1

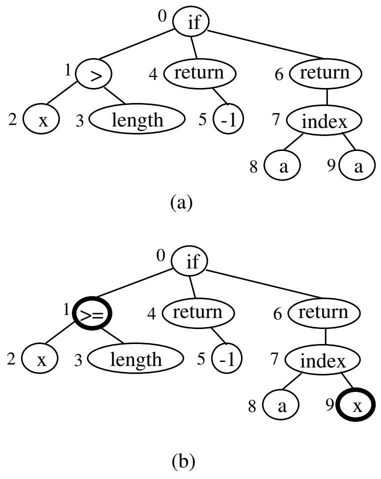

Figure 1: An AST before (a) and after (b) the application of the modification list: *mds* $= [(1, >=), (9, x)]$

Note, that the potential set of md_i to choose from is large. Even restricting ourselves to encoding the operator swap: ($>\rightarrow>=$), and swaps to variables, the exhaustive set of *mds*'s of length one for the AST in Figure 1(a) is:

$$\begin{aligned}\{\ & [(0,>=)], [(0,x)], [(0,\mathit{length})], [(0,a)],\\ & [(1,>=)], [(1,x)], [(1,\mathit{length})], [(1,a)],\\ & \ldots,\\ & [(9,>=)], [(9,x)], [(9,\mathit{length})], [(9,a)]\}\end{aligned}$$

most of these modifications cannot validly be applied to the AST – the search space is replete with invalid individuals. To limit the search space, pyEDB uses a genotype to phenotype mapping, partly inspired by that used in Grammatical Evolution[17], to ensure modifications are restricted to feasible locations. We describe this mapping next.

4.2 The Genotype to Phenotype Mapping

In pyEDB the modification list *mds* is the phenotype. The genotype g is a variable length bit-string. The key to the genotype to phenotype mapping is a small set of lookup-tables, called, *mod-tables*. These mod-tables are built during the initialisation of the evolutionary process. They encode the set of allowable changes to the AST P for a given type of modification. A mod-table takes the form:

$$\begin{aligned}MT_t = \{\ & (loc_1 \rightarrow \{targ_{11}, targ_{12}, \ldots, targ_{1n}\}),\\ & (loc_2 \rightarrow \{targ_{21}, targ_{22}, \ldots, targ_{2n}\}),\\ & \ldots\\ & (loc_m \rightarrow \{targ_{m1}, targ_{m2}, \ldots, targ_{mn}\})\}\end{aligned}$$

where the subscript t is a number denoting the type of modification handled by the table. In pyEDB: MT_1 handles modifications for relational operators: $>, \geq, <, \leq, ==$ and MT_2 handles modifications of variable names. Each row in a mod-table consists of a node address: loc_i and the set of target values $\{targ_{i1}, targ_{i2}, \ldots, targ_{in}\}$ that the node at loc_i in P can take during program modification.

During fitness evaluation, each genome is broken into 32-bit genes which are then used perform a staged-lookup of the mod-tables to produce a sequence of individual $(loc_i, targ_i)$ modifications.These *building* and subsequent *lookup* processes warrant further elaboration. We explain these in turn.

4.2.1 *Building the Mod-Tables*

The mod-tables are built prior to the evolutionary process with the assistance of structures called target-sets. Target sets are generalised rewrite rules encoding all of the modifications that it is possible to make to each value. An example of a target set is TS_1, the current set of valid operator swaps in pyEDB:

$$\begin{aligned}TS_1 = \{\ & (>\rightarrow \{>=, <, <=, ==\}),\\ & (>=\rightarrow \{>, <, <=, ==\}),\\ & (<\rightarrow \{>, >=, <=, ==\}),\\ & (<=\rightarrow \{>, <, <=, ==\})\}\end{aligned}$$

The variable target-set TS_2 maps each unique variable occurrence in the AST to the variables in scope of that occurrence. In our experiments TS_2 contains exactly the same entries as MT_2[3]. Given these TS_i we can build the corresponding MT_i by traversing the AST and looking up the appropriate entry in TS_i for each node[4].

To illustrate the process, a snapshot of the of building the mod-tables for the AST in Figure 1(a) is shown in Figure 2. This process works by traversing the AST P and using the target-sets to add entries to the mod-tables. The arrows in the figure represent corresponding values used for lookup

[3]Though this would of course change if we changed the variable substitution rules so that the set of valid rewrites was not dependent on node location.

[4]For general rewrite rules this process would include matching any indeterminates. But this step isn't needed in our examples.

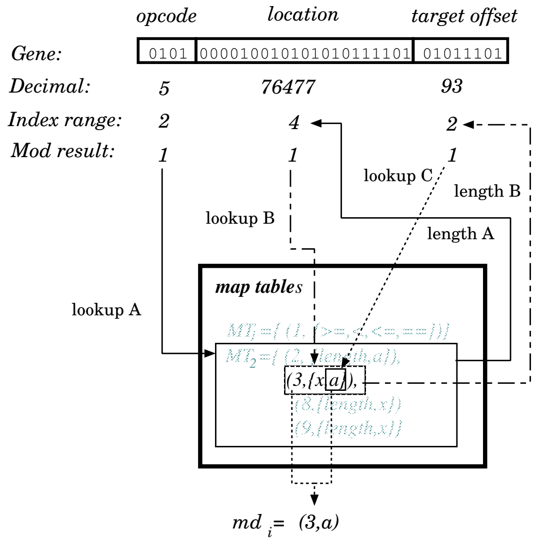

Figure 3: Three-stage lookup of the mod-table from the genome

and update. In the figure, P has already been traversed to node eight. The variable name corresponding to this node can be found in target-set TS_2 containing the possible target values each variable in P. The entry for node 8 maps to the set of target values $\{length, x\}$. This set, containing the other variables currently in scope of this access to a, is added to the mod-table MT_2 along with the node number[5].

Note, that each target-set in Figure 2 has different origins. TS_1 contains a fixed mapping supplied by the builders of pyEDB while TS_2 was populated by walking over P and collating other variables in the scope of each variable. This concludes the explanation of how the mod-tables are built next we explain how they are looked-up.

4.2.2 *Looking up the Mod-Tables*

As mentioned earlier, the genotype to phenotype mapping breaks the genome into a series of 32-bit genes each of which is used to code a single $(loc_i, targ_i)$ modification[6]. Figure 3 illustrates the process of looking up entries in the completed versions of the mod-tables from Figure 2 to create a single modification. The top part of the Figure shows the genome and the numerical values derived from it with the help of the mod-tables. The top row describes the gene itself. The first four bits are the opcode describing type of modification. The second field is a 20-bit value for locating the change in the AST. The last eight bits choose the target value for the modification. The lookup process has three stages: first, use the opcode to lookup the chosen mod-table; second, use the location to lookup the chosen location in the chosen mod-table; third, use the target-offset to lookup the chosen target value at the chosen location.

All of these lookups are done modulo the number of choices available so every gene will map to a valid modification with respect to the mod-tables. Before the lookup commences we convert each binary field to decimal. In the first stage the decimal value for the op-code (5 in this case) is modded with the number of mod-tables (two) to produce the first mod index (1 in this case). This index 1 denotes the second of the mod-tables (lookup A) yielding MT_2. Next, the length of MT_2 (4 in this case) is transmitted back (length A) as an index range for the next lookup. The next lookup is for the location to be modified, the decimal value 76477 is modded with 4 to also produce 1 which is used to index into MT_2 (lookup B). The indexed row of the lookup table $(3, \{x, a\})$ contains two target entries so the number 2 (length B) is transmitted back for the third lookup which proceeds analogously to isolate the final target value a. This value is then paired with the location to produce the modification: $(3, a)$ (i.e. replace the value at node 3 with "a").

This concludes our description of the genotype to phenotype mapping. This mapping produces only allowable changes[7]. There is scope for reducing the search space further by biasing the mapping using trace and contextual information.

4.3 Biasing the Mapping

The genotype to phenotype mapping described thus far is unbiased – for each operator every allowable change is equally likely. This is a good scheme if we don't have access to reliable information about *where* the defects might be and *what* the defects might be. However, if we do have such information then biasing modifications to the potential targets should reduce the search space.

In the following we describe, in turn, mechanisms we use to bias the search to the likely locations of defects (the where) and the likely cause of defects (the what). In pyEDB both biasing mechanisms can be switched on or off, at the request of the user, prior to an evolutionary run.

4.3.1 *Biasing to location*

Past work in evolutionary defect repair has used defect localisation algorithms to give a rough indication of where defects might reside[2, 11]. This has succeeded in reducing the search space thus making the defect repair more effective.

In this work we use a similar approach using the Tarantula defect localisation algorithm[12]. Tarantula counts the number of times each line of code is executed during both failing and passing tests. The counts are then used to produce a suspiciousness index S_i for each statement i in the program. Statements that are executed only during failed tests are regarded as highly suspicious. We implemented Tarantula for pyEDB to extract the S_i. pyEDB then runs through the following process to bias the lookup First, we share S_i for each statement among all the nodes in that statement giving a per-node S_j. Second, we normalise the sum of S_j for each mod-table so the S_j for each mod-table sums to one. Finally we add a cumulative index field to each map table entry being the sum of normalised S_j from the start of the table to that node.

The cumulative index fields described above, implicitly allocate each node a floating-point range that is proportional to its share of sum of S_j for its table. Now indexing into each mod-table, under the biased scheme, is then simply a matter of linearly mapping the location code in the gene to

[5]For our current variable substitution rules, the production of MT_2 from TS_2 is a simple line-by-line transcription.
[6]Any surplus bits at the end of the genome are ignored.

[7]This, of course says nothing about the correctness of the allowable changes, for instance it is easy to induce type-errors.

a value x between zero and one and then searching for the first entry with a cumulative index field above x. The nodes on the most suspicious lines will be hit more often.

4.3.2 *Biasing to error*

Biasing to error applies only to modifications of variable names. It is intended to make it easier to correct typographical errors. It is implemented by sorting the targets $\{targ_{i1}, \ldots, targ_{in}\}$ in each entry $(loc_i, \{targ_{i1}, \ldots, targ_{in}\})$ of the variable mod-table MT_2. The sorting is done by a measure of edit distance between the $targ_{ij}$ and the variable at loc_i, and is performed in two steps: first, we sort the target variables by string length; then, we perform a stable sort by *minEditDist*, the minimum edit distance between the variable at loc_i and the current target variable $targ_i$. These two sorting steps have the effect of favouring longer strings over shorter in the case of equal minimum edit distance. That is, we consider "count" and "counts" to be closer together than say "x" and "j".

Once the sorting of the targets is done, the target variable in scope which is typographically closest to the current is first in the target list. During evolution, the lookup is then done by generating an index: $index = rand()^{2.5} \times len(targs_i)$ where *rand()* randomly generates a number in the range $[0 \ldots 1)$, and $len(targs_i)$ is the length of the target list for the current variable. The use of the power 2.5 biases indexing toward the first elements.

4.4 Evolutionary Framework

For maximum flexibility we built a simple bespoke genetic algorithm (GA) framework in Python. The framework implements a generational GA and is configurable, allowing different forms and frequency of crossover, mutation and selection operations to be performed. The framework is set up to create the mod-tables and prepare for biasing operations prior to starting the first generation. All individuals are random bit-strings ranging in length from 32-bits – one gene – up to a user-specified maximum length. This maximum length trivially sets an upper-bound on the number of changes that can be encoded.

The first generation is seeded with individuals ranging linearly from 32-bits to to the maximum genome length. If a genome shrinks to less than 32-bits in length during the evolutionary process it is padded out to 32-bits with random values. Other settings of the algorithms are user-specified and we describe these along with our experimental results.

5. EXPERIMENTAL RESULTS

Our results fall into the categories of: **Effectiveness**, how well pyEDB fixes errors; **Efficiency** how well pyEDB finds errors; **Accuracy**,avoidance of over-fitting and absence of of spurious changes;**Sensitivity to Genome Length**; and **Effect of Biasing**. We present these in turn after the following outline of the experimental setup.

Except where stated otherwise, the results below were generated with the following settings and benchmarks. A cut and splice crossover was used, with each pair of parents yielding two offspring. Crossover can occur at any point on the genome[8]. Crossover is applied with a probability of 0.8. Mutation randomly inserts, deletes or replaces up to eight bits of the genome. Mutation applies to an individual with probability 0.3. We use tournament selection with a tournament size of four. We implemented elitism, preserving the single best individual in each generation. Except where noted, population size is 200.

```
def middleFunc(x,y,z):
    m = z
    if y < z:
        if x < y:
            m = y
        elif x < z:
            m = x
    else:
        if x > y:
            m = y
        elif x > z:
            m = x
    return m
```

Figure 4: The `middleFunc` benchmark

In each experiment pyEDB is given a program file containing defects and six to eight tests, most of which fail on the given program. pyEDB gives minimum fitness to any program that fails to compile, raises an exception or takes more time than a user-defined limit to complete. The evaluative function returns the number of test cases failed. A fitness of zero indicates that all test have passed. pyEDB is configured to subtract a small bonus (< 1.0), inversely proportional to the length of the genome, for *shorter* genomes that also pass all tests. Most tests were run until either an individual with an evaluative function score of zero or less was found or a maximum of 50 generations was reached[9]. Except where stated otherwise we used a maximum genome length of five corrections (160 bits) in our experiments.

Two benchmarks were used. A small benchmark, called `middleFunc` shown in Figure 4. And a larger benchmark, a publicly available python solution[10] to the facebook `smallWorld` puzzle challenge[10]. This program contains 61 lines, 130 variable accesses and complex conditional logic.

5.1 Effectiveness

To assess effectiveness we pyEDB against the two benchmarks. For the `middleFunc` benchmark we randomly added three variable and operator errors using pyEDB's modification framework to make ten different benchmark programs. We ran pyEDB 30 times on each of these ten benchmarks (with biases off) and counted number of generations each run took to either isolate an individual that passed all tests, or give up at 50 generations. Results varied widely with one, benchmark, tending to be solved after one generation, taking approximately 30 seconds on our desktop machine. In contrast the most difficult benchmark took an average of 22 generations (about 8 minutes) but this includes individual runs that gave up after 50 generations. On average the framework ran for an average 8.6 generations across all programs.

pyEDB is sensitive to the placement and nature of de-

[8]We attempted to force crossover points to 32-bit gene boundaries but this worked extremely poorly.

[9]50 is quite low but because we intend to refine pyEDB into a tool we need relatively fast runs.

[10]see: `http://cs.adelaide.edu.au/~brad/smallworld.py`

fects. This is perhaps unsurprising, because it is known that defects can hide each other[9]. This is especially the case with errors in relational operators which can redirect control away from some sections of code altogether. In the worst case, defects can be placed such that no correct output is produced unless all defects are fixed, which leads to a featureless fitness landscape.

For the larger `smallWorld` benchmark we ran fewer tests to measure effectiveness. We ran 10 trials on fixing a single defect, in this case a a few characters transposed in a randomly selected variable reference. Again we had biasing switched off. The resulting runs all isolated the error with a mean run of 6.6 generations (corresponding to about 10 minutes run-time) and a maximum run of 16 generations.

In order to test pyEDB on more defects we ran a small number of single experiments on `smallWorld` with two errors with minimum-edit distance bias on for variable swaps. For all of these were able to get results within 20 generations. However, searches for multiple typographical errors with search bias off and search for more than two errors in general failed to yield results after 50 generations.

In summary pyEDB can be effective in removing small numbers of operator and typographical errors from small benchmarks in reasonable time. However, effectiveness appears quite sensitive to the placement of errors. This is likely to be acceptable in its intended environment where it provides assistance to remove defects rather than blanket assurance of defect removal. Next we briefly assess the efficiency of pyEDB.

5.2 Efficiency

As a simple assessment of efficiency we compared the number of new individuals generated by pyEDB with the number of individuals generated for random search on the `middleFunc` benchmark. To implement random search we simply set the population size to 10000 and stopped searching after generation zero. This gives an upper bound of 10000 on the extent of our random search. Given this setup, using our genotype-to-phenotype encoding, a-priori a random individual has respectively, an approximate 1/850 and 1/9000 chance of fixing, two and three errors involving only *relational* operators in `middleFunc`. The corresponding figures are 1/31000 and 1/1500000 for two and three errors involving only incorrect *variable-names*. Our ten benchmarks involve different mixes of relational operator and variable name errors and we expect probabilities to range between these upper and lower bounds.

We ran trials for a `middleFunc` with two and three errors using both pyEDB and random search. Note we deliberately avoided testing the search for one error since we found that both pyEDB and random search would typically find the solution in generation zero, before any evolutionary operators had been applied. The position of the first individual to pass all tests was recorded as the search length. The results are summarised in Table 1 The results on the first three rows reflect the outcomes from 30 trials. The results on the last row reflect the outcomes of 20 trials (by which time the results were clear).

As can be seen pyEDB on two errors moderately outperforms random search both in terms of both average performance and reliability. For three errors the result is much stronger with the random search running out of time in most cases. In future, it would also be good to compare pyEDB with a hill-climbing algorithm to assess the relative effectiveness of the genetic operators in this context.

method	mean	stdev	median
pyEDB 2 errors	445.3	190.5	411
random 2 errors	1360.7	1487.9	891.5
pyEDB 3 errors	900.6	242.5	833.5
random 3 errors	9375.5	2103.2	10000

Table 1: Comparison of random search to pyEDB (number of individuals generated before finding a solution)

```
set $bestGenome = G$
for all $2^n$ possible sub-lists $s$ of $G$
    if $s$ is perfectly fit and shorter than $bestGenome$ then
        set $bestGenome = s$
end loop
return $bestGenome$
```

Figure 5: Algorithm for eliminating redundant modifications

5.3 Accuracy

The positive behaviours we assess in this section are the avoidance of overfitting, and the avoidance of spurious changes. Overfitting occurs when the algorithm specialised the program to the test cases but fails to produce a solution that generalises to other tests. In our trials, overfitting was rare. In *almost* all cases the individuals that passed all tests were identical to the original program without the defects[11]. At least in part, this is due to our choice of benchmarks: both the `middleFunc` and `smallWorld` are very sensitive to the perturbation of both relational operators and variable names – there is little scope for neutral changes.

We observed no spurious changes in any of the large number individuals we inspected. In any case, spurious changes can be weeded out by the very simple algorithm, shown in Figure 5 that can be applied to a perfectly fit genome $G = [g_0, \ldots, g_{n-1}]$ where each g_i is a gene encoding one modification. Through exhaustive search, this algorithm is guaranteed to find the shortest version of G that passes all the tests. Because, in practice,we limit G to a sensible length (always less than five here) this exhaustive search take less time than to run pyEDB for an extra generation.

5.4 Genome Length Settings

Programs can be quite sensitive to peturbation and pyEDB quite reliably encodes its genes into perturbations. This gives rise to the risk of deleterious genes undoing the good work of other parts of the genome. As a preliminary investigation into whether deleterious affected search we re-ran all the trials in the experiments from section 5.1 with different maximum genome lengths. If the presence of deleterious genes is a problem we would expect to see better performance with maximum genome lengths closer to number of repairs required. The results for the `middleFunc` bench-

[11]An interesting exception to this occurred when pyEDB found a way to engineer the `smallWorld` benchmark to cause tests to run zero times – giving perfect fitness!

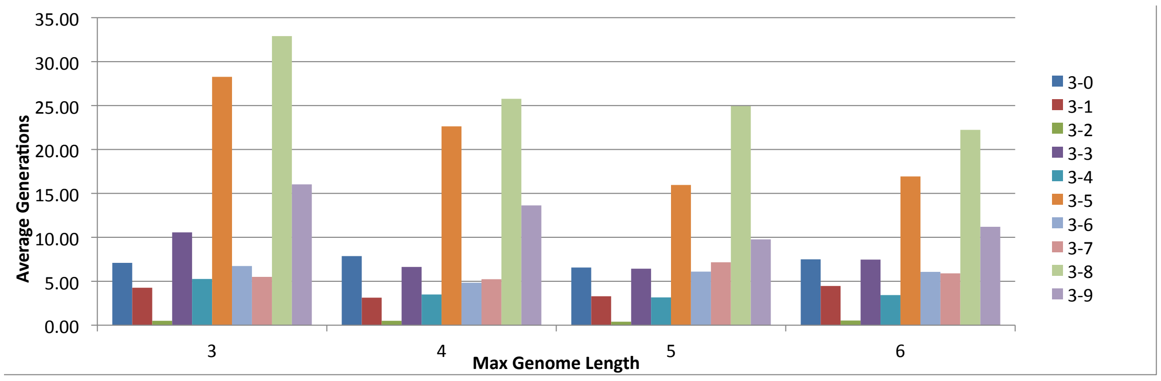

Figure 6: Time to correct benchmarks with varying maximum genome length for `middleFunc` benchmark

max changes	mean	stddev	median
max 1 change	5.4	4.2	4.0
max 2 changes	4.2	3.3	4.0
max 3 changes	6.6	5.0	8.0

Table 2: Effect of varying max genome length for `smallWorld` benchmark (number of generations used to generate correct solution)

max changes	mean	stddev	median
max 1 change	3.8	4.7	1.0
max 2 changes	3.0	4.0	2.0
max 3 changes	3.4	4.3	2.0

Table 3: Results of trials from Table 2 with minimum-edit-distance bias switched on.

mark are shown in Figure 6. These graphs show, that for all benchmarks the there is some benefit derived from setting the maximum allowable genome length to somewhat more than the minimum length required to make the change (this minimum is three changes in this case).

To verify these results extend to larger examples we re-ran the trials from section 5.1 on `smallWorld`. The results are shown in Table 2 Each row's results come from ten trials. Though these results need to be confirmed with larger numbers of trials, they indicate that there is some benefit for extending the maximum genome length beyond the minimum needed (in this case one gene) but this benefit tails off quite rapidly as the genome is made larger. This may indicate a stronger effect from deleterious genes for larger examples.

5.5 Biasing

As mentioned previously, there are two types of search bias available in pyEDB:minimum-edit-distance (MED) on strings to help choose *targets*, and tarantula weightings on *locations*. We present the results of our experiments from using these in turn.

To test the effect of bias in minimum edit distance we reran the `smallWorld` benchmarks from section 5.1 above on the same small typographical error. The results are shown in Table 3. These results seem indicative improvement. However, if the variable name error is not small, as may be the case if the programmer just uses the wrong variable, then the MED bias will work against you. The decision on using the MED bias depends on whether you are quickly searching for typographical errors or slowly searching for variable-substitution errors.

The Tarantula bias directs the search to locations where the error is more likely to be. We ran a number of preliminary experiments using this bias and found that, when the Tarantula data was accurate it was helpful in shortening the search time. However, in the presence of more than one defect the Tarantula data could be very inaccurate – for instance it is very easy switch relational operators in the `middleFunc` example so that Tarantula gives poor results on the rest of the code. In our experiments, especially those with errors in relational operators we found that Tarantula's error localisation was often misleading. This is likely to be, in part, a function of the benchmarks we are using and the fact that we concentrate on relational operators. In any case, we found that the worst-case behaviour of our benchmarks was better with defect localisation bias switched off. This concludes our review of our experimental results. Next we briefly review similar work on defect repair.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have described a new method for automatic defect repair based on evolving patches rather than programs. The results thus far show promise especially in terms of minimising the extent of the repair.

In future we would like to extend the framework with a greater variety of repair types. There is much scope for work to ascertain, across a broader range of benchmarks, whether it is generally useful to combine repair types or search for one type of repair at a time. There is also scope apply much more intelligence to the choice of useful candidates for variable swaps. We would also like to experiment with running fault localisation software incrementally as repairs are applied to improve its reliability. Finally, we would like to explore the prospect of building pyEDB into a continuous integration system as a background mechanism for suggesting repairs to code failing regression testing. If the brevity of corrections it produced here can be maintained more broadly it might prove a very useful tool.

7. REFERENCES

[1] A. Arcuri. On the automation of fixing software bugs. In *ICSE Companion*, pages 1003–1006, 2008.

[2] A. Arcuri. Evolutionary repair of faulty software. Technical report, University of Birmingham, May 2009.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *ICSE (1)*, pages 265–274, 2010.

[4] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using findbugs on production software. In *OOPSLA Companion*, pages 805–806, 2007.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[6] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[9] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *ISSRE*, pages 165–174, 2009.

[10] Facebook. It's a small world, January 2011.

[11] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. In *GECCO*, pages 947–954, 2009.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[13] J. R. Koza. Introduction to genetic programming tutorial: from the basics to human-competitive results. In *GECCO (Companion)*, pages 2137–2262, 2010.

[14] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[15] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[16] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, 2008.

[17] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Trans. Evolutionary Computation*, 5(4):349–358, 2001.

[18] H. D. Owens, B. F. Womack, and M. J. Gonzalez. Software error classification using purify. In *ICSM*, pages 104–113, 1996.

[19] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 3(6):41–49, 2005.

[20] M. Stumptner and F. Wotawa. A model-based approach to software debugging. In *In Proceedings on the Seventh International Workshop on Principles of Diagnosis, Val*, pages 214–223, 1996.

[21] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[22] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[23] J. L. Wilkerson and D. Tauritz. Coevolutionary automated software correction. In *GECCO*, pages 1391–1392, 2010.

[24] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.

[25] Y. Zhang and Y. Ding. Ctl model update for system modifications. *J. Artif. Intell. Res. (JAIR)*, 31:113–155, 2008.

[26] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.