Evolving Transformation Sequences using Genetic Algorithms

Deji Fatiregun, Mark Harman and Robert M. Hierons
Department of Information Systems and Computing,
Brunel University,
Uxbridge, Middlesex, UB8 3PH.
email {ayodeji.fatiregun, mark.harman, rob.hierons}@brunel.ac.uk

Abstract

Program transformation is useful in a number of applications including program comprehension, reverse engineering and compiler optimization. In all these applications, transformation algorithms are constructed by hand for each different transformation goal. Loosely speaking, a transformation algorithm defines a sequence of transformation steps to apply to a given program. It is notoriously hard to find good transformation sequences automatically, and so much (costly) human intervention is required.

This paper shows how search-based meta-heuristic algorithms can be used to automate, or partly automate the problem of finding good transformation sequences. In this case, the goal of transformation is to reduce program size, but the approach is sufficiently general that it can be used to optimize any source–code level metric. The search techniques used are random search (RS), hill climbing (HC) and genetic algorithms (GA).

The paper reports the result of initial experiments on small synthetic program transformation problems. The results are encouraging. They indicate that the genetic algorithm performs significantly better than either hill climbing or random search.

1 Introduction

Program transformation [2, 3, 7, 9, 11, 16] has been widely described as the changing of one program into another. It involves altering the program syntax while leaving its semantics unchanged. It has been extensively applied to various software engineering disciplines such as program synthesis, optimisations [1, 14], refactoring, reverse-engineering [15], program comprehension [4], software maintenance [17]. Transformation has also been shown to be a useful supporting technology for search–based software testing using evolutionary search techniques [10, 11]. Amorphous Slicing is a further application scenario where the source-to-source transformations proposed in this work may be applied.

In program transformation, we typically apply a number of small simple atomic transformation rules called axioms (some authors refers to them as transforms) to parts of a program's source code. These axioms are formally proven to be correct in that they are semantic equivalence preserving transformations.

Consequently, in the search for transformation sequences, we presume that if each axiom preserves semantic equivalence then a whole sequence of axioms ought to preserve semantic equivalence. Examples of simple transformation axioms (rules) are:

T_1: `x:= x + 1; x := x + 1;` $\Rightarrow$ `x := x + 2;`
T_2: `if (true) s1; else s2; fi;` $\Rightarrow$ `s1;`
T_3: `if (e1) s1; else s2;` $\Rightarrow$ `if (!e1) s2; else s1;`
T_4: `x := x + 1; y := y + 1;` $\Rightarrow$ `y := y + 1; x := x + 1;`
T_5: `for (s1; e2; s2) s3;` $\Rightarrow$ `s1; while (e2) s3; s2;`

Consider the program fragment shown in figure 1. We show the application of a sequence of transformations `[T4, T1, T1, T5, T2]` (given that the current program cursor position along the program parse tree is on line 1) to the fragment. and the resultant output program shown in figure 2.

Typically, with most transformations systems currently in use today, the order of application of the transforms

```
x := x+1;
y := y+1;
x := x+1;
IF (x>y) THEN
   a := a+1;
ELSE
   b := b+1;
FI;
a := a+1;
```

Figure 1: Source fragment: source program before sequence is applied

```
y := y+1;
x := x+2;
IF (x>y) THEN
   a := a+1;
ELSE
   b := b+1;
FI;
a := a+1;
```

Figure 2: Output fragment: resulting program from the application of sequence `[T4, T1, T1, T5, T2] to` $prog1$

are pre-determined by the designers of the transformation engine. This pre-fixing of the order of execution is obviously very dependent upon the particular input program and not generic and as such assumes *apriori* knowledge of the input program. The effectiveness of a given sequence is also dependent upon the order in which the transforms occur. That is, two sequences with identical axioms but with different placements in the sequence such as `[T1,T2,T3]` and `[T2,T1,T3]` could potentially produce different results for a given source program. Cooper et al. [5] describe this as interplay, where a transformation may create opportunities for other transformations and similarly, may eliminate opportunities.

We propose a system where we can dynamically generate transformation sequences for a variety of programs also using a variety of objective functions. In this paper, we consider optimising the program with respect to the size of the source–code i.e., the number of Lines of Code (LoC), where we aim to minimise the number of lines of code by as much as possible. For instance, in the example fragments 1 and 2, a sequence that produces Program $prog2$ is better than one that results in $prog1$ because it contains fewer nodes.

However, this approach is such that if we were interested in another singular metric or a combination of various metrics, then it would be relatively easy to include our new objective function into the algorithm.

An obvious approach to solving the problem would be to randomly traverse through the search space of possible sequences for a fixed number of iterations, keeping the best sequence found across each iteration. An exhaustive search of the search space is clearly infeasible due to there being an exponential number of combinations of axioms and their nodes of application.

The contributions of this paper are to:

- Reformulate the transformation problem as a search issue for optimisation.
- Provide evidence that evolutionary and/or local search can be used to evolve good transformation sequences.
- Investigate the difference between Hill-Climbing and the Genetic Algorithms. In summary, the GA outperforms the HC in every case.

The rest of the paper is organised as follows: Section 2 describes the rationale for treating the transformation problem as a search problem and the approaches taken. Section 3 describes the source transformations used in the study. Section 4 presents the experiments carried out and a discussion of the findings. Section 5 describes the related work in this area while section 6 outlines some of the outstanding issues for future consideration. Finally section 7 presents the conclusions.

2 Transformation as a Search Problem

An overall transformation of a program p to an improved version p' typically consists of many smaller transformation tactics [2, 3]. Each tactic consists of the application of a set of rules. A transformation rule is an atomic transformation capable of performing the simple alterations like those captured in examples $T_1 \dots T_5$. At each stage of the application of these rules, there are many points in a program; typically one per node of the Control Flow Graph. The set of pairs of possible transformation rules and their corresponding application point is therefore large.

Furthermore, to achieve an effective overall program transformation tactic, many rules may need to be applied, and each would have to be applied in the correct order to achieve the desired result. This explosion in the number of possible choice of transformation rules and their appropriate sequencing has led to problems in the generalisation of program transformation and while specialised transformation algorithms exists for dedicated tasks, we are currently unaware of any general purpose transformation algorithms.

Also, the search–space of possible transformation sequences make exhaustive search infeasible. Given a transformation system with 20 possible transforms and a sequence length of 20, there are 20^{20} possible combinations in the search–space which is huge. It is for these reasons that we employ evolutionary and meta–heuristic search algorithms to guide the search for good sequences and maintain that any technique which outperforms random search is perhaps useful.

2.1 Local Search - Hill–Climb

In a local search method such as the Hill–Climbing algorithm (HC), one guesses a solution within the solution space and then moves toward a better solution closer to the goal.

We implement a HC algorithm to find good transformation sequences. In our implementation, an initial sequence is generated randomly and serves as our starting point. The fitness of this individual is computed. The algorithm iterates through each neighbour to the current position and when a better individual is found, this individual replaces the old one as the current best individual. This process is repeated and if no better neighbour is found, we assume we have arrived at the top of the hill and the current solution remains our best.

The algorithm is restarted several times using a random sequence as the starting individual each time. The aim is that this would divert the algorithm from any local optima and increase the chances of finding a global solution.

2.2 Global Search - Genetic Algorithm

A Genetic Algorithm (GA) is a population-based search procedure, which starts with an initial random population and evolves over several generations, such that the individuals in successive generations have better or at least of no worse fitness values than those in preceding generations. An optimised individual is one which presents a more desirable solution to the given problem.

GAs imitate the natural process of evolution and use evolutionary operators such as crossover and mutation to alter the population across several generations toward optimality.

Crossover facilitates the exchange of information between two chromosomes by dividing each chromosome at a selected position and swapping adjacent sides across, creating new child chromosomes. Selection is the process by which the two individuals needed for crossover are chosen. The crossover rate is the probability of crossover execution. Mutation can allow the opportunity to introduce diversity into the population by allocation a chance for a gene to be altered. The probability that a gene within a chromosome would be altered is referred to as the mutation rate.

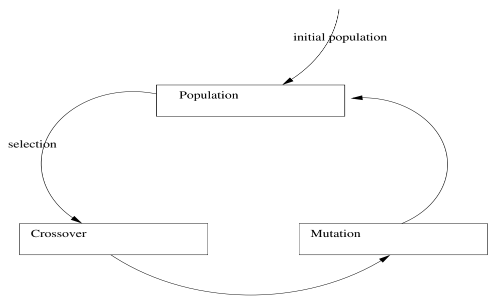


Figure 3: Iterative stages of a Genetic Algorithm, from the initial Population, crossover through to the application of mutation operators.

The result is a new population and the evolutionary operators are applied iteratively across each new population. The GA is terminated after a finite number of generations. Figure 3 summarises the iteration that occurs during the main stages of a GA.

We employ a GA to evolve the sequences of transforms that would result in the best possible program. Our approach is to use the transformation sequence to be applied to the program as the individual to be optimised. Using the transformation sequence as the individual makes it possible to define crossover relatively easily. In our implementation, we combine two sequences of transformations using a single point crossover, selecting a point at random along the chromosome and swapping adjacent sides. The result is a valid transformation sequence and since all transformation rules are meaning preserving, so are all sequences of transformation rules. Each time a parent sequence is applied to our source program, its fitness value is computed and likewise for every child sequence generated during crossover and re-admitted into the population space.

2.3 Fitness Function

We measure the fitness of a potential solution as the nominal difference in the lines of code between the source program and the new transformed program created by that particular sequence.

This is evaluated by:

1. compute oldLength = length of the input program.
2. generate transformation sequence (randomly, through crossover and mutation in the case of the GA, or next neighbour in the HC)
3. apply the transformation sequence to the top of input program.
4. compute newLength = the length of the program after sequence has been applied
5. compute the fitness of individual (oldLength - newLength)

2.4 Encoding

We refer to each transformation sequence as an individual which has a fixed sequence length of 20 possible transformations. In our implementation, we employ 22 different transformations numbered serially from 1. An example of an individual is:

```
[10,14,1,1,8,16,14,4,22,15,20,4,8,20,9,2,12,20,9,21].
```

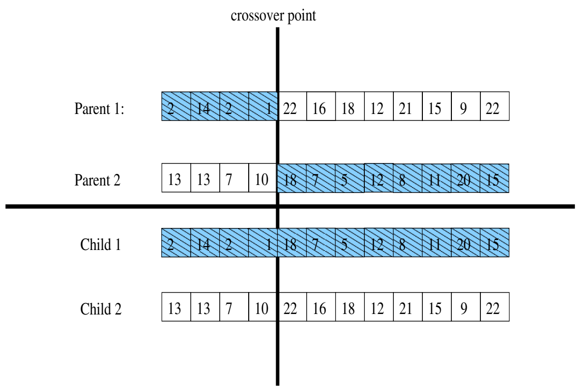


Figure 4: Gene sequence showing single point crossover of two parents creating two new children.

In our hill–climb implementation, we define the neighbour as the mutation of a single gene from the original sequence leaving the rest unchanged. The sequence

```
[10,14,1,18,16,14,...]
```

would therefore have as a possible neighbour

```
[9,14,1,18,16,14,..]
```

because the first gene would have been changed.

This search through each neighbour for a potentially better gene sequence is continued until no better sequence can be found. We restart the algorithm with a new random sequence.

This encoding of the GA makes it possible to define crossover relatively easily, we choose a random point and swap genes creating two new children whose fitness can then be evaluated. Figure 4 shows the process of crossover and the creation of the two new child individuals.

3 FermaT Transformations

We implement our algorithms using the FermaT transformation tool, which has a number of built-in transforms that could be applied directly to any point within the program.

The transformations in FermaT could, if their application is possible, either increase or decrease the size of the program. Some transformations may leave the program size unchanged. We do not include those transformations that could transform an entire program in one single step but rather atomic ones that work on pairs of nodes as previously shown in our examples.

In addition to transforms which alter the structure of the source programs in some way, we also include in the sequence, transforms such as @Right, @Left, @Up and @Down, that do not change the syntax of the input program but merely allow the movement of the current cursor position along the parse tree for the source program.

The inclusion of such transforms mean that we also need not statically determine the position of application for the optimising transforms but allow the algorithm to walk through the input program to determine when and where a valid transform should be applied. Every other transform apart from the four described above is a WSL-to-WSL transformation of the input source designed to alter the code in some way. In the experiments carried out, the following 20 transformations were used [1]:

`@double-to-single-loop:`	This transformation will delete all the 'COMMENT' statements within the selected code.
`@else-if-to-elsif:`	This transformation will replace an 'Else' clause which contains an 'If' statement with an 'Elsif' clause. The transformation can be selected with either the outer 'If' statement, or the 'Else' clause selected.
`@elsif-to-else-if:`	This transformation will replace an 'Elsif' clause in an 'If' statement with an 'Else' clause which itself contains an 'If' statement. The transformation can be selected with either the 'If' statement, or the 'Elsif' clause selected.
`@merge-right:`	This transformation will merge the selected statement into the statement that follows it.
`@merge-left:`	This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.
`@remove-redundant-vars:`	This transformation will remove any redundant variables in the source program
`@absorb-right:`	A transformation that would absorb into the current statement, the one that follows it.
`@absorb-left:`	A transformation that would absorb into the current statement, the one that precedes it.
`@simplify:`	This transformation would simplify any components as fully as possible
`@up:`	Moves up one level on the parse tree from the current item if possible.
`@down:`	Moves down one level on the parse tree from the current item if possible.
`@right:`	Moves right into next item on program parse tree
`@left:`	Moves left into the next item on program parse tree
`@move-to-right:`	This transformation will move the selected item to the right so that it is exchanged with the item that follows it.
`@move-to-left:`	This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.
`@add-left:`	This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.
`@delete-all-comments:`	This transformation will delete all the 'COMMENT' statements within the selected code.
`@combine-wheres:`	will combine two nested WHERES into a single structure
`@delete-all-redundants:`	deletes all redundant variables
`@delete-all-skips:`	This transformation will delete all the 'SKIP' statements within the selected code.

[1]Source for the description of FermaT transformations compiled from the transformation manual

A test for validity is carried out before each transformation is applied at the current cursor position. Each transformation is only performed if its application at that point is valid. If the test for the application of a transformation is invalid then it is not applied and the program remains unchanged.

4 Experiments

4.1 Experimental Setup

In this paper, we examine heuristic search methods in order to investigate the potential for finding good transformation sequences that optimise the size of source code. We define a fixed length of 20 genes for our transformation sequence for all three algorithms researched. We find that this value best represents the transformations sequence that would return the optimum program, with the programs included in our tests.

We implemented a standard Genetic Algorithm using single point crossover, a crossover rate of 100% and a mutation rate of 7%. We adopt the tournament selection technique for choosing the mating parents and create a single offspring from the mating which in-turn replaces the worse of the two parents in the population. We define a constant population size of 50 and run the algorithm over 200 generations.

We also conduct some analysis using a hill–climbing algorithm with multiple restarts, keeping the best individual over the different restarts. The HC algorithm uses a first–ascent technique, where the first better sequence found is our new current best position. We describe the notion of neighbourhood in our hill–climb as a the mutation of a single gene within an individual (keeping all other genes fixed). The algorithm is restarted 10 times with a new random individual each time.

We compare the results from both our GA and hill–climb with those returned by a purely Random Search of the search–space. The Algorithm for the random search randomly generates individuals and the best individual found so far is recorded across several iterations. The three algorithms were tested using synthetic programs with sequences, selections and iterations in order to get a feeling of how the transforms handled these.

We observe two outputs from the execution of the three algorithms: the most desirable sequence of transformations that it finds and the number of fitness evaluations that it takes to arrive at that solution. In the implementation for the algorithms, we keep the source program static, meaning that each new transformation sequence generated is applied to the same starting program as all other sequences before it. The effects of a previous sequence on the program are discarded before the next sequence is applied.

As a means of fair comparison, we only analyse the number of fitness evaluations it takes each algorithm to find individuals with good fitness score. Each algorithm is allocated a maximum of 3,000 fitness evaluations and recorded the best fitness recorded so far at intervals of 100 fitness evaluations. The results presented are averaged over 10 runs for each algorithm.

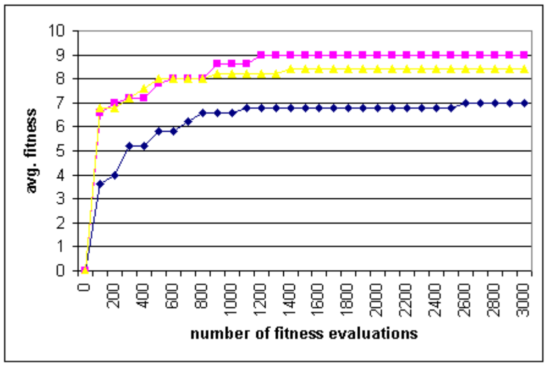
4.2 Results and Discussions

We find that perhaps unsurprisingly, the genetic algorithm outperforms both the random search and the hill–climber as the source program size increases. We observe in figures 5 and 6 that the gaps between the GA and the HC and Random search seem to increase across successive graphs. However also interestingly, we see that the random search outperforms the hill–climb algorithm for certain test programs. Each graph represents the results of executing the three algorithms on a single test program. The mean fitness value over 10 runs is plotted on the y-axis against the number of fitness evaluations on the x-axis.

It takes fewer number of fitness evaluations for the GA to match the fitness values for programs returned by the random search and the HC algorithm. The differences in the target programs from both algorithms were however not significantly different. We think this might be due to the synthetic nature of the test programs and are keen to try the experiment on benchmark programs for an experience of what the results might be.

We observe that the hill–climber repeatedly performed worse than RS over all tests. We feel that this is due to the implementation of the hill–climb chosen where we apply tight conditions for neighbourhood. We are also keen to try out scenarios where our neighbourhood criteria is randomised and more relaxed. We suggest that the hill–climber cannot perform any worse in such a scenario and also in one where the initial individual in the hill–climb cycle is perhaps the best one returned by the random search.

In the test cases where random search outperforms both the GA and the HC (figure 7), we observe that the sequences being generated by the GA and HC are not 'moving' toward areas where potential optimisations may be



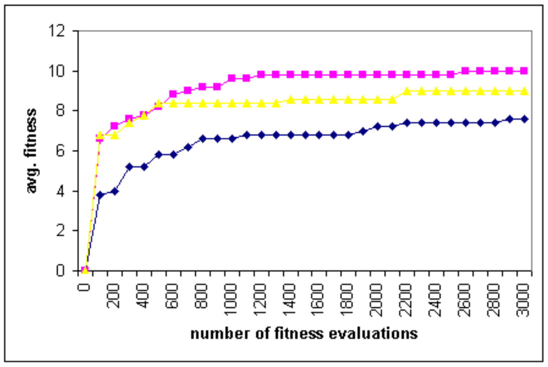
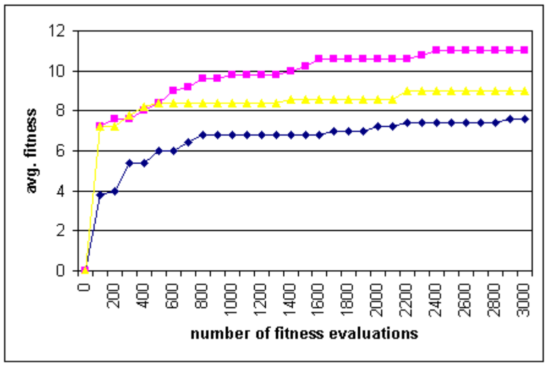


Figure 5: Increasingly improving GA performance over RS and HC. Successive test program requiring more advanced transformation sequences to produce high fitness values. The top line represents average fitness values for the GA, the middle line represents the values for Random Search and the bottom line represents the HC.



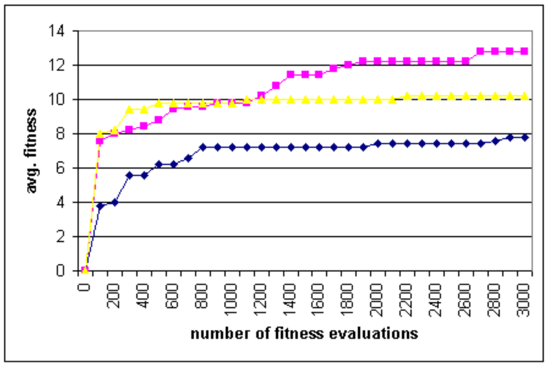


Figure 6: Continuing increase in GA performance over those of Random and Hill climbing. Top Line represents the average GA values, Middle line represents the average Random values while the bottom line represents the average HC values

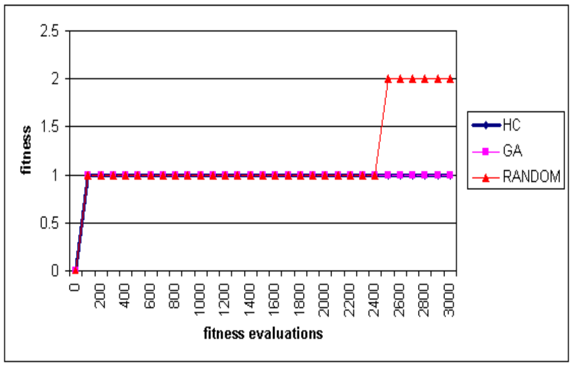


Figure 7: Tests on programs with IF-THEN-ELSE blocks

	GA v. Random	GA v. HC	Random v. HC
Test 1	.007	.011	.033
Test 2	.004	.005	.010
Test 3	.003	.004	.010
Test 4	.004	.005	.010
Test 5	.004	.004	.011
Test 6	.004	.005	.010
Test 7	.005	.005	.011

Figure 8: Results of Wilcoxon significance tests showing strong significance in the results with the GA outperforming both random search and the HC.

carried out, whereas the nature of the random search allows for a certain percentage of luck in arriving at good locations for optimisation.

Because we have no guarantee that the experimental data is normally distributed, we carried out the Wilcoxon's tests for significance of the results. This is a non-parametric test to test the significance of the difference between two samples. The results of the tests presented in Figure 8 shows strong significance in the results across various test cases.

In analysing our GA implementation, we believe that the GA potentially kills off good subsequences of transformation during crossover. There is a need to preserve these subsequences whenever crossover occurs to improve the overall quality of solutions. Finally, we suggest is that there may be certain features intrinsic to some programs that may make random search not so bad against the others. Examples of our test programs include a sequence of assignments such as those shown in Figure 9

For the example in Figure 9, program one, a good solution would be a sequence of six absorb-right transformations that would result in the following program:

```
x := x + 7;
```

Similarly, for programs two, a sequence of two `absorb-right`, three `move-right` and then more `absorb-right` transforms would produces the following optimal programs:

```
a := a + 1;
b := b + 1;
c := c + 1;
x := x + 7;
```

while for program three, a sequence of two `absorb-right` transforms and a `down` transform would allow further simplifications possible inside the IF-block. Further `absorb-right` transforms may then be applied. A good result would be a sequence that produces the following output:

```
x := x + 3;
IF x > y THEN
  b := b + 2;
  x := x + 1;
ELSE
  c := c + 2;
FI;
x := x + 3;
```

5 Related Work

Some prior work has been done in the area of using meta–heuristic search algorithms to search for optimisation sequences. Cooper *et* al. [5, 6] focus on searching for sequences of compiler optimisation transforms which work largely on compiled code using biased random sampling. They compare the results of their experiments with those

program one	program two	program three
`1. x := x + 1;`	`1. x := x + 1;`	`1. x := x + 1;`
`2. x := x + 1;`	`2. x := x + 1;`	`2. x := x + 1;`
`3. x := x + 1;`	`3. x := x + 1;`	`3. x := x + 1;`
`4. x := x + 1;`	`4. a := a + 1;`	`4. IF x > y THEN`
`5. x := x + 1;`	`5. b := b + 1;`	`5.    b := b + 1;`
`6. x := x + 1;`	`6. c := c + 1;`	`6.    b := b + 1;`
`7. x := x + 1;`	`7. x := x + 1;`	`7.    x := x + 1;`
	`8. x := x + 1;`	`8. ELSE`
	`9. x := x + 1;`	`9.    c := c + 1;`
	`10. x := x + 1;`	`10.    c := c + 1;`
		`11. FI;`
		`12. x := x + 1;`
		`13. x := x + 1;`
		`14. x := x + 1;`

Figure 9: Sample test programs, showing the increasing number of moves to the right that a good sequence would require to produce an optimal solution. Program one requires a sequence of absorb transformations to produce the optimal result. In order to simplify program two and three, the algorithms need to be intuitive to realise where potential optimisations may be applied, e.g., in program three, the program cursor needs to go DOWN the Abstract Syntax Tree (AST) for the program, into the IF-block for further simplifications to occur.

obtained against a fixed set of optimisations in a predetermined order. Fatiregun *et* al. [8] conducted preliminary investigation into search–based transformations. Ryan [13] worked on using search techniques to automate parallelisation for supercomputers. Nisbet [12] focused on using a GA to find program restructuring transformations for FORTRAN programs to execute on parallel architectures. Zou and Kontogiannis [17] have carried out research on transforming procedural legacy codes to object-oriented codes using the Markov model and the Viterbi algorithm to identify an optimal sequence of transformations.

Our work continues in this area of merging search and transformation problems, but specifically applied to transformations performed at source–code level. We do not predetermine the order of our sequences but compare different techniques for best results. The correct ordering of good transformation sequences would be as much a goal for our search algorithms.

6 Future Work

Currently, research into this problem has introduced a number of issues that ought to be tackled in the immediate future. We describe the need for a more responsive fitness measure that rewards moves to potentially good transformations. This problem of the evolutionary search not adequately moving to potentially good locations may be improved by an improvement in the fitness functions being used. Presently, the fitness measure is rather too fine grained where a better individual (sequence) is one which *definitely* makes the new program size smaller than the source program.

However, this technique punishes those transformations that may not necessarily make the program smaller but may take the cursor to a point where a potentially good optimisation may be exploited. This is especially important because: Firstly, the order of the transformation sequence is essential and secondly, because it provides the basis for the evolution of good building blocks or sub-sequences.

We also advocate for the adoption of hybrid search, to further enhance the performances of the meta–heuristic and evolutionary searches. For instance, we could seed our hill–climb with the best individuals found from the execution of either the random search or GA. Similarly, we could perhaps also seed the initial GA population with a top percentile of the results found by either the hill–climb and / or random, before applying the GA operators.

Lastly, another direction for future research is looking at dynamic programs, where the input program changes if a fitter individual is found. This in essence allows us to create arbitrarily long sequences that potentially could result in optimal results for any objective function.

7 Conclusion

This paper highlights the transformation problem and describes search–based approaches to solving this problem. We look at using evolutionary and meta-heuristic search algorithms, in our case, Genetic Algorithms and Hill–Climbing algorithms to traverse a large search space of individuals to find a good transformation sequence that minimises the number of lines of source code. We find that these techniques are useful for the problem described and should be further explored and that the GA outperforms the HC algorithm in all test cases examined.

8 Acknowledgements

The authors would like to thank Martin Ward for use for the FermaT transformation engine in this work and discussions on search-based transformations. The authors would also like to thank Dr. Stephen Swift for discussions on search-based transformations. They would also like to thank Kiarash Mahdahvi for providing help with the statistical analysis.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[2] I. D. Baxter. Transformation systems: Domain-oriented component and implementation knowledge. In *Proceedings of the Ninth Workshop on Institutionalizing Software Reuse*, Austin, TX, USA, Jan. 1999.

[3] K. H. Bennett. Do program transformations help reverse engineering? In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 247–254, Bethesda, Maryland, USA, Nov. 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[4] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[5] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimising for reduced code space using genetic algorithms. In *Proceedings of the 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, May 1999.

[6] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimising compilers for the 21^{st} century. *Journal of Super Computing*, 2002.

[7] J. Darlington and R. M. Burstall. A tranformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[8] D. Fatiregun, M. Harman, and R. Hierons. Search based transformations. In *Genetic and Evolutionary Computation Conference*, pages 2511 – 2512, Chicago, USA, July 2003. Springer.

[9] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, Jan. 1982.

[10] M. Harman, C. Fox, R. M. Hierons, L. Hu, S. Danicic, and J. Wegener. Vada: A transformation-based system for variable dependence analysis. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, Montreal, Canada, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. Voted best paper by attendees.

[11] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[12] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *HPCN Europe*, pages 987–989, 1998.

[13] C. Ryan. *Automatic re-engineering of software using genetic programming*. Kluwer Academic Publishers, 2000.

[14] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the International Conference on Functional Programming (ICFP'98)*, Baltimore, USA, September 1998.

[15] M. Ward. Reverse engineering through formal transformation. *The Computer Journal*, 37(5), 1994.

[16] M. Ward. Assembler to C migration using the FermaT transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)*, Oxford, UK, Aug. 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[17] Y. Zou and K. Kontogiannis. Migration to object oriented platforms: A state tranformation approach. In *Proceedings of the IEEE International Conference on Software Maintenance, (ICSM 02)*, pages 530 – 539, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.