A Multi–Objective Approach To Search–Based Test Data Generation

Mark Harman
King's College, CREST
Strand, London
WC2R 2LS, UK
mark.harman@kcl.ac.uk

Kiran Lakhotia
King's College, CREST
Strand, London
WC2R 2LS, UK
kiran.lakhotia@kcl.ac.uk

Phil McMinn
University of Sheffield
211 Portobello St, Sheffield
S1 4DP, UK
p.mcminn@dcs.shef.ac.uk

ABSTRACT

There has been a considerable body of work on search–based test data generation for branch coverage. However, hitherto, there has been no work on multi–objective branch coverage. In many scenarios a single–objective formulation is unrealistic; testers will want to find test sets that meet several objectives simultaneously in order to maximize the value obtained from the inherently expensive process of running the test cases and examining the output they produce.

This paper introduces multi–objective branch coverage. The paper presents results from a case study of the twin objectives of branch coverage and dynamic memory consumption for both real and synthetic programs. Several multi–objective evolutionary algorithms are applied. The results show that multi–objective evolutionary algorithms are suitable for this problem, and illustrates the way in which a Pareto optimal search can yield insights into the trade–offs between the two simultaneous objectives.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Algorithms, Experimentation, Measurement, Performance.

Keywords: Automated test data generation, evolutionary testing, multi–objective genetic algorithms

1. INTRODUCTION

Generating test data by hand is tedious, expensive and error–prone. Nonetheless, for many organizations this activity cannot be avoided, because adequate levels of testing are increasingly required or recommended by internationally accepted standards [7, 26] for quality assurance and safety. This combination of problem difficulty and importance has made automated test data generation a widely studied topic and one for which Search–Based Software Engineering techniques have proved to be particularly useful [4, 5, 16, 17, 19, 25, 28, 29, 31, 32].

This paper is concerned with the problem of structural test data generation and, in particular, the problem of generating branch adequate test sets. A test set is branch adequate if and only if, for each reachable branch b of the program's Control Flow Graph, there exists a test case that causes execution to traverse b. Branch adequate testing is recommended by the British Computer Society standard on testing [7] and is mandatory for avionics standards [26].

The problem of generating such a branch adequate test set has been formulated by several previous authors as a set of search problems, in which the fitness function measures the distance between the path traversed by a candidate test case, and a path required to execute the branch of interest [4, 5, 16, 19, 25, 29]. Each branch is considered in turn, yielding a separate search for each branch that has yet to be covered. This branch becomes the new test goal and a search is performed, guided by a fitness function that assesses proximity of the execution trace to this branch.

In all previous work on search–based branch adequate test data generation, the problem has been formulated as a single–objective search problem[1]; the sole objective is to cover the branch in question. While this is valuable, in many situations the tester may have additional goals that they would like to achieve using the same test set. For example, the tester may also wish to find test cases that are more likely to be fault–revealing, or test cases that combine different non–subsuming coverage based criteria. The tester might also be concerned with test cases that exercise the usage of the stack or the heap, potentially revealing problems with the stack size or with memory leaks and heap allocation problems. There may also be additional domain–specific goals the tester would like to achieve, for instance, exercising the tables of a database in a certain way, or causing certain implementation states to be reached.

In any such scenario in which the tester has additional goals over and above branch coverage, existing approaches represent an over simplification of the problem in hand. A multi–objective optimization approach would be a more realistic approach. The lack of any previous work on multi–objective test data generation therefore highlights an important gap in the literature. This paper takes a first step towards filling this gap. The paper formulates the test data generation problem as a multi–objective optimization problem, presenting results from a study of a Pareto optimal approach and a weighted fitness approach.

[1]Hermadi and Ahmed [12] propose a multi–objective approach whereby they try to find the minimal set of test cases which cover a set of target paths, but do not use a Pareto–based approach.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'07, July 7–11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-697-4/07/0007 ...$5.00.

In order to scope the problem, it was necessary to consider a suitable multi–objective scenario. As the previous discussion indicates, there are many choices. The scenario adopted is one in which the tester wishes to achieve branch coverage, while also constructing test cases that exercise the dynamic memory allocation of the program under test. This scenario would occur where, for example, the tester knows that memory is highly constrained or where the tester believes that there may be memory leaks or possible null pointer dereferences.

The paper presents results from five case studies of this multi–objective problem when applied to real code from the Software–artifact Infrastructure Repository [1] and also to specially constructed examples that denote extreme cases where the two objectives are either in full agreement or total opposition. These extreme cases allow the approaches to be explored at the limits for which they might be expected to be applied. The primary contributions of the paper are as follows:

1. The paper introduces the first formulation of test data generation as a multi–objective problem. It describes the particular goal oriented nature of the coverage criterion, showing how it presents interesting algorithmic design challenges when combined with the non goal oriented memory consumption criterion.

2. The paper presents results that confirm that multi–objective search algorithms can be used to address the problem, by applying the 'sanity check' that search–based approaches outperform a simple multi–objective random search.

3. The paper presents results that suggest that a suitably constructed weighted multi–objective approach, though simplistic, can be effective for this problem in some cases. However, there is also evidence that a Pareto optimal approach can find better solutions in other cases, with respect to a certain objective. The paper shows how a weighted and Pareto hybrid approach can be used to complement each other.

4. The paper also presents results from the application of a Pareto optimal evolutionary algorithm, assessing the impact of the interdependencies that arise between the set of search problems denoted by the set of branches to be covered.

The rest of this paper is organised as follows. Section 2 provides an overview of background information on multi–objective evolutionary algorithms. Section 3 describes two different approaches for attempting branch coverage while maximizing dynamic memory allocation. It also outlines the NSGA-II algorithm by Deb et al. [8] used in one approach. Sections 4 and 5 present the experimental setup and five case studies comparing both approaches when applied to synthetic and real world programs. Insights gained during the studies are revealed in Section 6. Section 7 presents related work and Section 8 concludes.

2. BACKGROUND

Multi–Objective Evolutionary Algorithms (MOEAs) are algorithms designed for solving problems where no single optimal solution exists and a set of solutions is required instead. An example of a Mutli–Objective Problem (MOP) is the Knapsack problem [18], where weight has to be minimized and profit maximized. This is also a typical example where two objectives are in direct conflict with each other.

As stated, MOPs require a set of solutions known as a Pareto optimal set. Such a set contains only non–dominating solutions. The concept of domination is defined as

> Individual X dominates Y if, and only if, X is better than Y in at least one objective, and no worse in all other objectives.

A Pareto front and Pareto optimal set can be defined as ([13])

> **Pareto Optimal Set:** For a given MOP $\vec{f}(\vec{x})$, the Pareto Optimal Set ($P*$) is defined as: $P* := \{\vec{x} \in F \mid \neg\exists\vec{x'} \in F \ \vec{f}(\vec{x'}) \leq \vec{f}(\vec{x})\}$, where F is the decision variable space.

Genetic Algorithms (GAs) are naturally suited for MOPs because they maintain a number of individuals in every population, each of which may be better suited for one objective than another. Another benefit is that a GA can optimize multiple objectives in parallel. A Pareto GA exploits this feature, giving it the ability to generate a Pareto optimal set in a single run.

A special type of single–objective GA, such as a weighted GA, might achieve the same results over a number of runs. However, for each run, the objectives have to be formulated as a set of constraints rather than optimization problems.

Most MOPs, such as the Knapsack problem mentioned above, have a large set of 'good compromise' solutions compared to the set of 'unpractical' solutions. For example, the only 'useless' solutions for the Knapsack problem are the zero weight or minimal profit solutions, *i.e.* the very extreme points on a Pareto front.

The opposite is true for the MOP addressed in this paper. Branch coverage is attempted at a subgoal level, with each goal corresponding to a branch in a program. It follows that branch coverage can only have a boolean outcome; either a test case traverses the branch or it fails to do so. The latter test cases are of no practical use, therefore branch coverage has to be achieved, even if this should mean failing to allocate any memory. Hence, the set of desirable solutions is very restricted; it may only contain one solution.

This observation would suggest that a weighted GA is better suited for this problem. However, as memory allocation may be optimizable for some branches (*e.g.* by increasing loop–iterations which allocate memory), a Pareto GA might also be adequate, especially since finding the right set of 'weights' can prove challenging and a different set of weights might be required for each branch.

In a weighted GA, each objective is given a coefficient, acting as a 'weight' for its fitness value. The fitness values for each objective are then combined into a single value, from which point onwards the multi–objective GA becomes identical to a single–objective GA. In order for a weighted GA to be effective, particularly when two objectives are in conflict, the objectives have to be ordered or prioritized in some way. This is in contrast to a MOEA, which treats all objectives as equally important.

For the problem considered in this paper, one objective, maximizing memory allocation, is not clearly definable or quantifiable, because memory consumption may not have an obvious 'optimum'. Giving it too much weighting might

inhibit branch coverage, because of the adverse effect on the overall fitness value (depending on the code/inputs). On the other hand, excessively reducing its weighting might render it insignificant. The aim of maximizing memory allocation would thus be reduced to a random search.

3. IMPLEMENTATION

The IGUANA tool [21] was adapted for the implementation of the two algorithms, which are based on a model described by Wegener et al. [29]. In addition to the standard *branch distance* computed for a test case with respect to a branching condition, Wegener's model uses an *approach level*. It reflects the distance between branching nodes in a Control–Flow Graph and a target and is aimed at helping the GA find the 'quickest' path to a target by incorporating path information into the fitness function.

In order to measure memory allocation, the source code was instrumented with a global variable, used to count all bytes allocated during the execution of a function. The freeing of memory by dereferencing a pointer was not accounted for in this study, because all the memory was allocated to global pointers which were released outside the scope of the function under test. The next section describes the operations used by the GAs and their configuration.

Two different types of selection operators were used for the implementation of the weighted and Pareto GA. For the Pareto approach an elitist selection and reinsertion strategy was chosen. Elitism ensures that the current best individual (or a set of best individuals in case of a multi–objective GA) is copied across into the next generation. The weighted GA uses stochastic universal sampling [2] as a selection method, where the probability of an individual being selected is proportionate to its fitness value. This means 'fitter' individuals have more chance of being selected, but an 'unfit' individual may still be included, thereby partially maintaining diversity within the population in order to prevent a premature convergence at a sub–optimal solution.

Before individuals are selected for crossover, they must be ranked according to their fitness value within the population. The weighted-GA uses linear ranking [30] with a selection pressure Z of 1.7 (in accordance with the Wegener model), where ordered individuals are assigned fitness values such that the best individual has a fitness of Z, the median individual a fitness of 1.0 and the worst individual a fitness of $2 - Z$, where $Z = [1.0, 2.0]$. These ranked values are then converted into proportionate fitness values before selection takes place. Ranked fitness values for the Pareto GA are calculated based on the Pareto-ranking method described in Section 3.1.

Discrete recombination [23] was used to produce offspring and the mutation algorithm is based on the breeder genetic algorithm [23]. It defines a mutation probability of $1/len$, where len is the length of the input vector. Each of the breeding populations contains a different mutation step size p, ranging from 0.1 to 0.000001. A mutation range r_i is defined for each input parameter x_i by the product of p and the domain size of x_i, with $0 \leq i < len$. The 'mutated' value v_i of x_i can thus be computed as $v_i = x_i \pm r_i \cdot \delta$. Addition or subtraction is chosen with an equal probability. The value of δ is defined to be $\sum_{x=0}^{15} \alpha_x \cdot 2^{-x}$, where each α_x is 1 with a probability of 1/16 else 0. If v_i is outside the allowed bounds of x_i, its value is set to either the minimum or maximum value for x_i.

A competition manager controls the number of individuals each population evolves. Every 20 generations, 10% of the individuals are randomly chosen for migration from one population to another. A migration manager ensures a population will only receive individuals from at most one other population. The competition manager also calculates a progress value for each population at the end of a generation. This progress value p is computed for a population at generation g as follows: $0.9 \cdot p + 0.1 \cdot rank$. $rank$ indicates the average fitness of a population and is obtained by linearly ranking the individuals within a population, as well as the populations amongst themselves. Again, a selection pressure of 1.7 was used for obtaining each rank value.

Every n number of generations, where n is configured via the competition manager, the populations are ranked according to their progress values. After this, a reallocation parameter is computed for each population, which controls how many individuals from the worst performing populations are transfered to the best performing ones. However, a population is not allowed to loose its last five individuals to prevent it from dying out. The transfer of individuals between populations is aimed at improving the overall performance of the GA. Thus, in effect, the best performing population is seeded every n number of generations and, as a result, will contribute most to the number of fitness evaluations.

3.1 Pareto GA Implementation

The implementation adapted for this paper is based on the NSGA-II algorithm described by Deb et al. [8]. The main difference between a standard GA and a multi–objective GA is the way fitness values are computed and individuals ranked within a population. The NSGA-II algorithm creates a set of front lines, each front containing only non–dominating solutions. Within a front, individuals are rewarded for being 'spread out'. The algorithm also ensures that the lowest ranked individual of a front still has a better fitness value than the highest ranked individual of the next front.

The remainder of this section explains the ranking algorithm's steps in detail.

The first stage of the algorithm calculates two entities for each solution [8]; 1) a count c for the number of individuals which dominate the current individual; 2) a set of individuals which are dominated by the current individual.

All individuals with a count of zero, *i.e.* those not dominated by any other individual, are grouped together to form the first front. The individuals from this front are then iterated and for each individual in their 'domination set', the count is reduced by one. Individuals that subsequently end up with a count of zero are again grouped together to form the next front. This process is repeated until all individuals are assigned to a front.

In order to encourage diversity within a front and prevent premature convergence, individuals are rewarded for lying either at extreme ends or less crowded regions of a front. This is done by assigning a 'distance' attribute d to each individual, which measures the distance of an individual to its closest neighbours. The value of d is defined to be $\sum_{x=0}^{n-1} \delta_x$ where n is the number of objectives and δ_x the combined distance of an individual from its closest neighbours with respect to the current objective.

Thus, if two individuals have the same non–domination count c, the individual with the greater 'distance' d ranks higher.

3.2 Weighted GA Implementation

Unlike the Pareto GA, a weighted GA can only find a single best solution. This approach is commonly applied to multi–objective problems where it is possible to prioritize or order objectives in a meaningful way.

Branch coverage is a minimization task, where an ideal solution has an objective value of zero. In order to combine the objective value for the memory allocation with the distance measures, the inverse of the normalized number of bytes allocated was used. As this value can never reach zero, a cut–off point of 10^{-5} was chosen as the 'ideal' memory value and thus ideal overall fitness. If a branch fails to allocate any memory, a worst case value of 1000 was used as the objective value for the memory allocation.

Thus, if a test case allocates memory, the formula used to obtain the objective value is: $1.001^{-b} \cdot w_b^{-1} + 1.0 \cdot d$, where b is the *raw* number of bytes allocated, w_b is the weight for the memory objective, with $0 < w_b \leq 10^5$, and d the distance measure composed of *branch distance* and *approach level*. The weight for d was left constant at 1.0.

4. EXPERIMENTAL SETUP

Five case studies were carried out into the effectiveness of different search methods in generating branch adequate test data while maximizing dynamic memory allocation. The three searches considered are a random, Pareto optimal and a weighted search. Two case studies are based on real world C code and three on synthetic programs. The synthetic programs were chosen to evaluate the performance of a search in the context of 'extreme' examples. Although very small with respect to lines of code, the input domain for the synthetic programs ranges up to 10^{10}. The degree of difficulty for search–based testing is determined by the size of the search space as well as the shape of the fitness landscape, ensuring the synthetic examples are not trivial. In addition, the dynamic memory allocation was designed to add further complexity to the problem.

A search was terminated if either

1. an ideal solution was found, or
2. 100,000 fitness evaluations had been performed, or
3. no progress, with respect to the current best solution, had been made over 25 generations.

For the Pareto GA, an ideal solution was considered to be the best solution with respect to memory allocation, which also achieved the branch target.

5. CASE STUDY RESULTS

Each Case Study consists of the three algorithms run 10 times. The results of the case studies are presented in Figure 1.

Case Study 1 is the `addscan` function from the `space` program, used by the European Space Agency for scanning star field patterns. Memory is allocated without releasing it, thus to avoid memory leaks, the function was modified for test purposes to free the allocted memory.

Summary: Overall the Pareto GA allocates 66% more bytes than the weighted GA at the expense of branch coverage. The function contains 32 branches and its domain size is approximately 10^{539}. The weighted GA is deliberately directed towards covering branches, making it more likely to succeed within the limits set by the stopping conditions described above. The 3 branches left uncovered were either infeasible or the search simply failed for these.

In order for the Pareto GA to cover a branch it needs to find an 'extreme' point on the Pareto front. These points will only be 'discovered' quite late in the search, because a Pareto GA always tries to find a good spread of solutions across the front, evolving from a central region towards the 'end points' of a front–line.

Case Study 2 is a function taken from the `cgi-util.c` source, which is based on `post-query.c` and `query.c` by NCSA. It takes a string and a 'stop' character as input, and parses the string until either a terminating null or a stop character is found. The new substring is removed from the input string and returned by the function. Both parameters of the function have a direct impact on the amount of memory being allocated. However, this amount is constant for all branches because the memory allocation occurs at the start of the function. The length of the input string was restricted to 10^4 characters for practical reasons. This function also had to be modified to release the memory allocated after each test run to prevent memory leaks.

Summary: The results from Figure 1 confirm that a random search is good at achieving high branch coverage for 'easy to cover' branches. However, over 10 runs the search only manages to allocate 33% of the optimum for memory allocation. The weighted GA used a weight distribution of 3:2 in favour of branch coverage for this Case Study. It manages to beat the Pareto GA in 1 run, allocating the maximum amount of memory possible. However, over 10 runs the weighted GA allocates slightly less (263 bytes) than the Pareto GA and only manages to cover 100% of the branches during 1 run, compared to a 100% coverage achieved by the Pareto GA in 7 runs. Thus, the Pareto GA can be considered better suited for this Case Study, because achieving full branch coverage is required.

Case Study 3 is a function containing four predicates. The first three follow an `if()else if()` structure and allocate a constant amount of memory. The `true` branch of the first predicate allocates 20 bytes, the `true` branch of the second predicate 10 bytes and the third predicate, 5 bytes. The last predicate does not allocate any memory. This example was chosen to investigate an inverse relationship between 'approach level' for branch coverage and memory allocation and the effects on finding a Pareto optimal set.

Summary: The random search is uninteresting as it neither achieves 100% coverage nor allocates any memory at all in 9 of the 10 runs. The Pareto GA finds the maximum amount of bytes that can be allocated in 50% of the runs. However, it also leaves at best 3 branches uncovered. By achieving 100% branch coverage, the weighted GA manages to optimize the input vector to allocate the maximum amount of memory possible in all runs, clearly outperforming the Pareto GA at less computational cost.

Case Study 4 is a program that generates a random sequence of characters from the alphabet and stores them in a string. The length of the string generated depends on the input parameters. The first parameter affects the path to

be taken through the function; the second parameter specifies the length of the string. The branching nodes ensure that the second parameter only influences the memory allocation if the test case traverses the `true` branch of the first predicate. If a test case fails to cover this branch, the memory allocation for the rest of the function will either be constant, or, in one case, the function will exit prematurely and no memory will be allocated.

Summary: The input domain for Case Study 4 is 10^5 and the maximum number of bytes that can be allocated is restricted to 256. Even though the input domain is quite small, all methods fail to cover 100% of the branches. One of the uncovered branches is controlled by a predicate checking if the memory allocation was successful. The `true` branch of this predicate can be considered infeasible because of the restrictions imposed on the memory allocation. All other branches are covered by the weighted GA 40% of the time. Both the Pareto GA and random search fail to cover at least 1 more branch than the weighted GA. One of these uncovered branches is controlled by a flag–containing predicate. Flags are known to inhibit the performance of a GA [3] and flag controlled branches are unlikely to be covered by a random search. The success of the weighted GA in covering this branch can be explained by the distribution of weights. These ensure more resource is spent on the branch coverage objective compared to the Pareto GA, which shares its resources amongst the objectives.

Case Study 5 is a function constructed to produce a 'difficult to search' fitness landscape for the GAs [22]. To further add complexity, the memory allocation is constant for all but one branch. The `false` branch of a 'hard to cover' predicate contains an additional reallocation of memory, which in effect 'rewards' a search for missing the target, creating a deliberate conflict between the two objectives.

Summary: The random search fails to allocate more than 33% of the total possible memory and only covers 17% of all branches. Perhaps surprisingly the Pareto GA outperforms the weighted GA by covering at least 1 more branch than the weighted GA in all but 1 run. It also consistently allocates the maximum number of bytes possible over the 10 runs, whereas the weighted GA only manages to do so in 50% of the runs.

6. DISCUSSION

This section discusses the findings of the case studies from Section 5 and the results presented in Figure 1. It also includes some insights gained during the case studies.

For Case Studies 3 and 4 the weighted GA clearly outperforms the Pareto GA in all objectives. Case Study 1 presents a trade–off between the two GAs. The weighted GA covers an average 88% of the branches, compared to just under 80% covered by the Pareto GA. However, the weighted GA only manages to allocate about 34% of the number of bytes the Pareto GA allocates over 10 runs. In Case Study 2 the Pareto GA achieves an overall higher branch coverage than the weighted GA, while allocating an equal number of bytes. Finally, in Case Study 5 the Pareto GA again beats the weighted GA over a total of 10 runs.

The maximum number of bytes recorded during the case studies refer to the highest values found by a test case which covered a particular target. Any 'better' value found for this objective whilst attempting the target was not recorded if the test case missed the target and covered another branch, *e.g.* branch b instead. However, Figure 2 shows that the ideal solution for branch b will allocate at least the same amount of memory.

Overall, the findings suggest that it is not possible to pick one search method over another, as each performs better in some cases. For example, Table 1 illustrates that both the weighted and Pareto GA, cover branches missed by either Pareto or weighted GA respectively. Equally, both approaches have a number of disadvantages; the high computational cost associated with the Pareto GA and the difficulty of finding the most efficient set of weights for the weighted GA. For some example functions even a slight increase in the weight for the memory objective resulted in a significant drop in branch coverage. For others, adjusting the weights did not seem to affect the branch coverage and only marginally improved the memory allocation, even with a 'drastic' redistribution of weights. A set of experiments were carried out to investigate the impact different weights have on the behaviour of the weighted GA, but space prohibits a full discussion of these results.

Given the results presented in Table 1 and Figure 1, a hybrid approach may be advisable. For example, a weighted GA could be used to cover branches for which the Pareto GA failed to find test cases. This would also solve the issue of not being able to evaluate the performance of a weighted GA for an objective with an undefined optimum. Case Study 1 is an example where the maximum number of bytes allocated by the weighted GA is meaningless without a point of reference. The Pareto GA is more likely to find a good approximation to the 'real' optimum because it has less room for error, *e.g.* by not having an ideal distribution of weights.

The case studies also revealed that, in most cases, the Pareto GA does not produce a front–line. It converges at a single solution instead (see Figure 3). Where a front–line exists, it often lacks diversity. While this is not entirely due to the subgoal approach, it is emphasized by it.

For a front–line to contain many points, the target branch needs to either allocate varying amounts of memory, or an inverse relationship between the distance of a test case from the target and the memory allocated by it must exist (see Figure 2).

In any other scenario the front–line contains at most two points; one point representing a case that reached the target and the other, any test case that happens to allocate more memory than the first but misses the target.

Another issue revealed is that 100% branch coverage is very hard to achieve for programs containing `malloc/calloc/realloc` statements. After allocating memory, a well written program should check whether the allocation has been successful. It is these cases that are of interest, and which partly motivated the exploration of applying a Pareto GA to the branch coverage problem. However, to exhaust a program's heap space can be very challenging, possibly requiring a large number of loop–iterations or the presence of a memory leak to name but a few scenarios. Once the heap space is exhausted, the C program may crash, especially if it is not well written. As a result, the test environment will also terminate, thus being unable to log the test case that caused the crash.

7. RELATED WORK

The first research in this area used symbolic execution [15] and constraint solving [9, 24]. More recently, search–

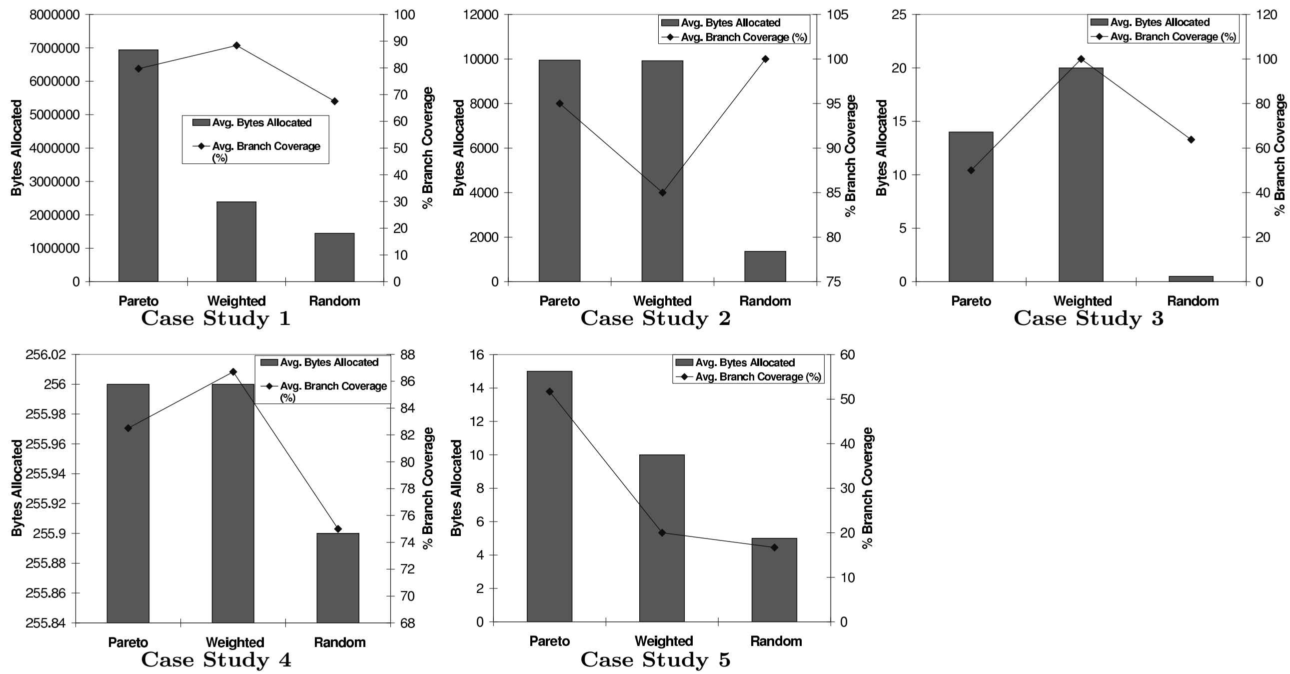

Figure 1: Results of the branch coverage and memory allocation achieved by three different algorithms: a random search, a Pareto GA and weighted GA

```
...
int p*;
if( a == 0)
{
   /*target 1T*/
}
else
{
   /*target 1F*/
   p = (int*)malloc(a*sizeof(int));
}
...
```

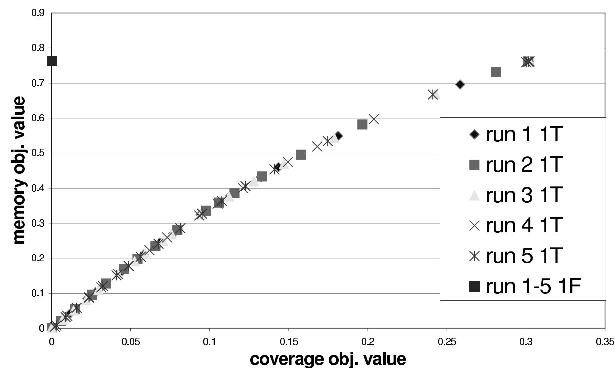

Figure 2: Final Pareto fronts produced for targets 1T and 1F. The upper point on the y–axis represents the 'ideal' solution for target 1F. As can be seen, once the branch has been reached, a single solution will dominate all others because it is the only branch allocating memory. When attempting to cover target 1T on the other hand, the Pareto optimal set potentially consists of an infinite number of solutions. The graph combines five runs which reveal little variance between the front–lines produced. Interestingly, the 'ideal' point for target 1F corresponds to the maximum value contained within the Pareto optimal set for target 1T with respect to memory allocation.

based approaches to test data generation have proved to be a popular application of Search–Based Software Engineering.

Many test data generation scenarios can be attacked using a search–based approach, with examples in the literature including stress testing [6], finite state machine testing [10] and exception testing [27].

The present paper is concerned with the problem of generating test data for structural testing; in particular branch coverage. Hitherto, this has turned out to be by far the most popular of all the applications of search–based test data generation. Korel [16] was one of the first authors to apply search–based techniques to the problem of branch adequate test data generation. Korel used a variation of hill climbing called the alternating variable method. Like other authors, for Korel the goal was a single–objective; to cover some 'difficult' branch not yet covered by a more lightweight random search. Xanthakis et al. [31] were the first authors to apply evolutionary computation algorithms to test data generation problems. They formulated the problem as a single–objective function of achieving coverage of paths. McMinn [20] provides a comprehensive survey of search–based test data generation.

In the past decade many authors have also addressed the problem of automated search for branch adequate test sets

Branch ID	Example Function
	`char *makeword(char *line, char stop) {`
	`int x = 0,y;`
	`char *word;`
	`word=(char *)malloc(sizeof(char)*(strlen(line)+ 1));`
1T/1F	`for(x=0;((line[x]) && (line[x] != stop));x++)`
	`word[x] = line[x];`
	`word[x] = '\0';`
2T/2F	`if(line[x]) ++x;`
	`y=0;`
3T/3F	`while(line[y++] = line[x++]);`
	`return word;`
	`}`

```
char *makeword(char *line, char stop) {
  int x = 0,y;
  char *word;
  word=(char *)malloc(sizeof(char)*(strlen(line)+ 1));

  for(x=0;((line[x]) && (line[x] != stop));x++)
    word[x] = line[x];

  word[x] = '\0';

  if(line[x]) ++x;
    y=0;

  while(line[y++] = line[x++]);

  return word;
}
```

branch ID	bytes allocated
1T	{9634}
1F	{9856}
2T	{9692}
2F	{1194}
3T	{9553}
3F	{9649}
Pareto optimal set	

Figure 3: The table to the right presents the Pareto optimal sets for each 'subgoal' of the example function used in Case Study 2. It combines the results collected over five runs and illustrates that it is often not possible to generate a Pareto front–line when considering branch coverage and memory allocation as a MOP. Although the amount of dynamic memory allocated depends on the input parameters, it is constant for all branches. As a result one solution will dominate all others with respect to a particular target.

branch ID	bytes allocated Pareto/weighted	distance Pareto/weighted
7	2826560/690360	0.001997004/0
9	2494888/534160	0.005979056/0
10	5622848/1529968	1.001997004/0
11	5978016/1309352	2.001997004/0
12	6372608/586784	2.001997004/0
15	6969776/518320	0.000699161/0
16	1374560/1304160	0.5/0
17	2644048/758032	1.000999001/0
21	2455024/195888	3.000999001/0
28	6728128/291368	0/6.2

Table 1: The table shows the branches covered by the weighted GA and not the Pareto GA, or vice versa. The 'distance' measure illustrates how close the best solution came to traversing the target branch. It combines the normalized branch distance and the approach level. A 0 distance indicates a branch has been covered. These results were obtained during Case Study 1.

[5, 14, 16, 17, 19, 25, 28, 29, 31, 32]. For example, Baresel et al. and Bottaci consider the problems of fitness function definition [4, 5]. Jones et al. [14], McGraw et al. [19], Pargas and Harrold [25] and Wegener et al. [29] introduce approaches to evolutionary search for branch adequate test data.

Other authors address closely related structural test adequacy criteria. For example, Xiao et al. [32] compare Evolutionary Testing with simulated annealing for the problem of condition-decision coverage, an alternative structural test data generation goal which can be reformulated as a branch adequacy problem using a testability transformation [11]. Mansour and Salame [17] consider the problem of path coverage, which is a stronger form of test adequacy than branch coverage, comparing evolutionary testing, hill climbing and simulated annealing.

However, despite the large body of work on search–based test data generation, all previous work has considered the problem as a single–objective problem. The present paper is the first to introduce a multi–objective formulation of the problem, considering both weighted and Pareto formulations of multi–objective optimality for the structural adequacy criterion of branch coverage.

Many other non–structural test data generation goals have been considered in the literature [6, 10, 27], and these have also been formulated as single–objective search problems. These other forms of search–based test data generation may also benefit from a multi–objective approach, as there may be several goals which the tester would like to achieve in determining a set of test data. However, the consideration of multi–objective formulations of non–structural test data generation remains a problem for future work.

8. CONCLUSION AND FUTURE WORK

This paper has presented a first multi–objective approach to branch coverage. Traditionally, the aim of branch coverage has solely been to find test cases which traverse a specific branch. The paper supplements this goal with the additional objective of consuming as much dynamic memory as possible at the same time.

Five case studies, two based on real world C code and three created to push the techniques to the extremes, compared the performance of three search methods: a random search, Pareto GA and weighted GA. The results show that a weighted GA is best suited in most cases, achieving the same results as a Pareto GA more efficiently. However, the studies also reveal that a hybrid approach between the two algorithms may offer the best overall results.

One of the issues revealed during the case studies was the balancing of weights between the different objectives. For example, the ratios used for the case studies presented in Section 5 were obtained by 'trial and error'. Future work will investigate if a meta–heuristic search method could be used to find an ideal set of weights. An empirical evaluation

on larger real world programs is also planned, comparing various algorithms applied to multi–objective branch coverage problems. For this, additional techniques, such as a hierarchical GA, will be considered.

9. ACKNOWLEDGMENTS

We would like to thank Afshin Mansouri for his advice on multi–objective algorithms. Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest) and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London. Kiran Lakhotia is funded by EU grant IST-33472 (EvoTest).

10. REFERENCES

[1] The Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.html.

[2] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Publishers, 1987.

[3] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 108–118. ACM, 2004.

[4] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[5] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[6] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In Hans-Georg Beyer and Una-May O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028. ACM, 2005.

[7] British Standards Institute. BS 7925-1 vocabulary of terms in software testing, 1998.

[8] Kalyanmoy Deb, Samir Agrawal, Amrit Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. KanGAL report 200001, Indian Institute of Technology, Kanpur, India, 2000.

[9] Richard A DeMillo and A Jefferson Offutt. Experimental results from an automatic test generator. *ACM Transactions of Software Engineering and Methodology*, 2(2):109–127, March 1993.

[10] Karnig Derderian, Robert Hierons, Mark Harman, and Qiang Guo. Automated Unique Input Output sequence generation for conformance testing of FSMs. *The Computer Journal*, 49(3):331–344, 2006.

[11] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[12] Irman Hermadi and Moataz Ahmed. Genetic algorithm based test data generator. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 85–91, Canberra, 8-12 December 2003. IEEE Press.

[13] Arturo Hernández Aguirre, Salvador Botello Rionda, Carlos A. Coello Coello, Giovanni Lizárraga Lizárraga, and Efrén Mezura Montes. Handling Constraints using Multiobjective Optimization Concepts. *International Journal for Numerical Methods in Engineering*, 59(15):1989–2017, April 2004.

[14] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[15] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[16] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[17] N. Mansour and M. Salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–134, 2004.

[18] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, New York, 1990.

[19] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[20] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[21] P. McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. *Technical Report, Department of Computer Science, University of Sheffield*, 2007.

[22] Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop*, pages 165–182, September 2005.

[23] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[24] A. Jefferson Offutt. An integrated system for automatically generating test data. In Raymond T. Ng, Peter A.; Ramamoorthy, C.V.; Seifert, Laurence C.; Yeh, editor, *Proceedings of the First International Conference on Systems Integration*, pages 694–701, Morristown, NJ, April 1990. IEEE Computer Society Press.

[25] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[26] Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992.

[27] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA 98)*, pages 73–81, March 1998.

[28] H.-C. Wang and B. Jeng. Structural testing using memetic algorithm. In *Proceedings of the Second Taiwan Conference on Software Engineering*, Taipei, Taiwan, 2006.

[29] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[30] Darrell Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation. In James D. Schaffer, editor, *Proc. of the Third Int. Conf. on Genetic Algorithms*, pages 116–121, San Mateo, CA, 1989. Morgan Kaufmann.

[31] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.

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