Locating dependence structures using search based slicing

Tao Jiang, Nicolas Gold, Mark Harman, Zheng Li

King's College London, Strand, London, WC2R 2LS, UK

Abstract

This paper introduces an approach to locating dependence structures in a program by searching the space of the powerset of the set of all possible program slices. The paper formulates this problem as a search based software engineering problem. To evaluate the approach, the paper introduces an instance of a search based slicing problem concerned with locating sets of slices that decompose a program into a set of covering slices that minimize inter-slice overlap. The paper reports the result of an empirical study of algorithm performance and result-similarity for hill climbing, genetic, random search and greedy algorithms applied to a set of 12 C programs.

Key words: Program Slicing, Search Based Software Engineering

1 Introduction

Dependence analysis has been applied to several stages of the software engineering process, such as program restructuring [21,53], program comprehension [26], regression testing [12] and program integration [42]. It can also be an effective way of understanding the dependence structure of a program [13,52] and a measurement of dependence-related attributes such as cohesion and coupling [10,60]. For these applications, sets of slices are used to reveal interesting properties of the program under analysis, such as the presence of dependence clusters and the cohesive (and less cohesive) parts of the program.

The advent of commercial, scalable and robust tools for slicing such as Grammatech's CodeSurfer [36] makes it possible to construct all possible slices for large programs in reasonable time. By constructing the set of all slices of a program, it is possible to analyse the dependence structure of the program. This allows slicing to be used to capture the dependence of every point in the program, allowing analysis of the whole program dependence structure. This raises an interesting research question:

"How can useful interesting dependence structures be formed in amongst the mass of dependence information available?"

In this paper, dependence is analysed using program slicing, and so this question is reformulated as:

"Of the set of all possible slices of a program, which subsets reveal interesting dependence structures?"

Of course, for a program consisting of n program points, there will be n possible slices and, therefore, 2^n subsets of slices. Since the number of program points is always at least as large as the number of statements in the program, the powerset of all possible slices will be extremely large; too large to enumerate for any realistically sized program. This is merely a reflection of the mass of dependence information available and would need to be considered by any whole program dependence analysis. The overwhelming quantity of information motivates the search based approach introduced in this paper.

The paper introduces an approach to location of dependence structures, founded on the principles of search-based software engineering (SBSE) [23,40]. Using this formulation, the problem becomes one of a search for a set of slices that exhibit interesting dependence structures. The choice of what constitutes an 'interesting dependence structure' is a parameter to the overall approach, making it highly flexible. In search based software engineering, a fitness function is defined to capture such a property of interest. In the case of search based slicing, it captures the properties of a dependence structure that make it interesting to a particular analysis.

The search process is realized by an algorithm that uses the fitness function to guide a search that seeks to find optimal or near optimal solutions with respect to the fitness function. In order to experiment with the search based slicing approach, the paper presents the results of an implementation and associated empirical study into the search for slice sets that decompose a program into a set of slices that cover the program with minimal overlap. The fitness function used in the empirical study is motivated by work on slicing as a decomposition technique [34,73].

This instantiation of the search based slicing approach formulates the decomposition problem as a set cover problem [31]. However, it must be stressed that this represents merely the *instantiation* of a parameter to the approach (the fitness function). The search based slicing approach derives a great deal of flexibility from the fact that the fitness function (and therefore the property of interest) is merely a parameter; in order to search for a different kind of dependence structure, only the fitness function needs to be changed.

The paper reports the results of experiments with four different search algorithms for search based slicing and presents the results of an empirical study involving 12 C programs. The empirical study aims to answer four related research questions:

(1) How well does each algorithm perform?
(2) How similar are the results produced by each algorithm?
(3) How can the results be visualized and what do they reveal?
(4) How efficiently can the best algorithm perform with large practical programs and for all the functions in programs?

The paper makes the following primary contributions:

(1) An approach that identifies dependence structures is introduced as a search problem over the powerset of the set of all possible program slices, allowing search based algorithms to be used to search for interesting dependence structures.
(2) A fitness function is introduced that seeks to optimise the search towards solutions that decompose the program into a set of slices that collectively cover the whole program with minimal overlap. Four search algorithms are implemented in order to experiment with this fitness function.
(3) The results of an empirical study are reported, showing that the greedy algorithm performs better than random, hill climbing and genetic algorithm approaches to the problem. This is an attractive finding, since greedy algorithms are extremely simple and efficient.
(4) A simple visualization is introduced to explore the results and their similarity. This shows a higher degree of similarity for the intelligent techniques over random search. This visual impression is augmented by computational analysis of results. The similarity of results for intelligent search provides that the results are consistent and meaningful.
(5) The visualization also has an interesting side effect, which may be a useful spin off: the presence of code clones becomes visually striking in some of the examples. However, clone detection is not the focus of this paper.
(6) The paper also reports results on redundancy. That is how often a slice is completely included by another one. The results suggest that redundancy phenomena are universal in 12 programs. However, it is shown that this redundancy does not affect the Greedy algorithm advocated in the paper.
(7) Based upon the performance comparison with 4 search algorithms, the greedy algorithm is further applied to 6 larger programs to decompose each function of each program. The results show that majority of functions can be decomposed into sets of slices efficiently.

The data used in this paper are made available to the research community to facilitate replication at http://www.dcs.kcl.ac.uk/pg/jiangtao/.

The rest of the paper is organised as follows: Section 2 presents the problem description in more detail, while Section 3 introduces the search-based algorithms and their application to the problem. Section 4 and 5 presents the results of the empirical study. Sections 6 and 7 present related work and conclusions.

2 Problem Description

The goal is to identify dependence structures by searching the space of all subsets of program slices. In this paper, static backward slicing is used, but the approach is not confined merely to static backward slicing; it can be used with any analysis that returns a set of program points (thereby including all forms of program slicing).

As an illustrative example, consider a program that has only 8 program points. Table 1 gives all the slices of this hypothetical example in terms of each program point as slicing criteria.

Program Slicing	Program point							
	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	0	1	1	1	1	0	0	0
3	1	0	1	1	0	0	0	0
4	0	1	1	1	1	0	0	0
5	0	1	0	1	1	0	0	0
6	1	1	1	1	1	1	1	0
7	0	0	1	0	0	1	1	1
8	1	0	0	0	1	1	1	1

Table 1
An example of looking for optimum properties in program slicing sets.

The table represents the value of each slice. In this table, a 1 represents a program point that is included in the slice, while a 0 represents a program point that is not included in the slice. In this situation, a good decomposition would be the set {1,5,7}, rather than {1,2,7}, {6} or any other subsets. The solution {1,5,7} is preferable, even though {1,2,7} has the same coverage as {1,5,7}, because {1,2,7} has more overlap than {1,5,7}; even though {6} has the same overlap as {1,5,7}, because {6} has less coverage than {1,5,7}. The other subsets have the same situation as the set {1,2,7} and/or {6}.

However, with increasing program size, the number of possible solutions grows exponentially. Therefore, the paper formulates this kind of slice subset identification problem as an optimization problems within the framework of Search Based Software Engineering (SBSE). SBSE is a framework for considering the application of metaheuristic search techniques to software engineering problems. The SBSE framework allows search based techniques to be used to provide acceptable solutions in situations where perfect solutions are either theoretically impossible or practically infeasible [40].

In order to apply the framework to a specific software engineering problem, it is necessary to reformulate the problem as a search problem [35,78,79]. This can be achieved by defining the search space, representation, and fitness function that describe the problem. The next 3 subsections explain each of these attributes of the formulation in more detail.

2.1 Search Space

The purpose of all search algorithms is to locate the best (or an acceptably good) solution among a number of possible solutions in the search space. The process of looking for a solution is equivalent to that of looking for some extreme value—minimum or maximum, in the search space.

In the experiments reported upon here, the search space is the set of all the possible sets of slices. Following Horwitz et al. [44], a 'possible slicing criterion' is taken to mean 'any node of the System Dependence Graph (SDG) of the program'. Therefore, for a program with n nodes in the SDG, there will be n corresponding slicing criteria and, therefore, 2^n subsets of slicing criteria. This space of 2^n subsets of slicing criteria forms the search space for this problem. Clearly, enumeration will not be possible since n can be arbitrarily large. This observation motivates the search based software engineering approach advocated in this paper.

2.2 Representation of Slicing

The representation of a candidate solution is critical to shaping the nature of the search problem. Frequently used representations include floating point numbers and binary code. In this problem, the representation of solutions is binary. The definition of representation of slicing can be formulated as a simple 2-dimensional array: Let $A[i, j]$ be a binary bit, i be a program point and j be a slicing criterion, so that $A[i, j] = 1$ if the slice based on criterion j includes the program point i and $A[i, j] = 0$ if the slice based on criterion j does not include the program point i. In this way, the array A denotes the set of slices of the program, with both array bounds determined by the number of program points (i.e. nodes of the SDG).

2.3 Fitness Function

The choice of a fitness function depends upon the properties of the set of slices for which the search algorithm will optimize. This choice is a *parameter* to the overall approach to search based slicing. In order to illustrate the search based slicing approach, this section introduces several metrics that will be used as fitness functions to decompose a program into a set of slices that collectively cover the entire program, while minimizing the degree of overlap between the slices.

These metrics are inspired by previous work on sliced-based metrics by Bieman, Ott and Weiser [10,56,60,65–69,75,76]. The following notation will be used.

Let M be the number of program points of the program, P be the number of program points of the optimal slicing set, $\cap(S_1, ..., S_i)$ be the intersection of i slices, $\cup(S_1, ..., S_i)$ be the union of i slices and $Max(S_1, ..., S_i)$ be the largest slice selected from i slices. All the metrics defined below are normalized. Normalization allows for comparison of metrics from differently sized programs, while the expression as a percentage is merely a convenience: the metrics are so-defined that 100% denotes the maximum possible value. The metrics used are as follows:

Coverage. This measures how much the program points in a slicing set cover the program points of the whole program. This metric was introduced by Weiser [75].

$$100 \cdot \frac{\cup(S_1, ..., S_P)}{\cup(S_1, ..., S_M)} \qquad 1 < P < M$$

Overlap. This Evaluates the number of program points of the intersection within a slicing set. It can be defined in many ways; this paper considers two possibilities:

Average For each pair of slices in the set, evaluate the percentage of program points that are in both. The average value is evaluated based on all such pairwise comparisons.

$$100 \cdot (\Sigma_{i=1}^{P-1} \Sigma_{j=i+1}^{P} \frac{\cap(S_i, S_j)}{Max(S_i, S_j)}) \qquad 0 < i < P$$

Maximum For each pair of slices in the set, evaluate the percentage of program points that are in both. The maximum value is the largest value among all pairwise comparisons.

$$100 \cdot Max(\frac{\cap(S_i, S_j)}{Max(S_i, S_j)}) \qquad 0 < i \neq j < P$$

With any definition of properties of interest, a mechanism is needed to map properties onto overall fitness values. For multiple objective problems, one simple technique for combining values for n fitness values: $Property_1, \ldots, Property_n$ is to combine them into a single fitness value using corresponding 'weights' $K_1, \ldots, K_n$.

In the experiments reported upon here, two fitness functions are defined, implemented and experimented with (corresponding to the two choices for measurement of average):

$$Coverage \cdot 0.5 + (100 - Average) \cdot 0.5 \qquad (1)$$
$$Coverage \cdot 0.5 + (100 - Maximum) \cdot 0.5 \qquad (2)$$

In both cases the weights are set to 0.5 so that each of the two objectives of the two fitness functions is considered equal. Nevertheless, decision of the weights is optional, different weights for the *coverage* and *overlap* could be considered in terms of the specific needs. As an illustrated example of fitness here, equal weights are considered since there are no other evidence that the *coverage* is more dominant to the *overlap* and vice versa. Both formulations of fitness attempt to capture the decomposition of the program that maximises coverage while minimizing overlap. Future work will consider the variation of these weights and the exploration of the Pareto front of optimal solutions.

3 Search Algorithms

This section describes the 4 types of search algorithms used in the experiments reported upon in the paper. The detailed description of these is given in algorithmic pseudo code in Figures 1, 2, 3 and 4.

3.1 Genetic Algorithm

A Genetic Algorithm (GA) [78] begins with a set of solutions (represented by chromosomes) called a population. Solutions from one population are used to form a new population. This is motivated by a hope that the new population will be better (according to the fitness function) than the old one. Solutions

```
Genetic Algorithm:
Parameters: Population(P): 50; Generation(G): 100; Crossover Probability: 0.8; Mutation Probability: 0.01.
Begin
  $i \leftarrow 0$
  while($i < G$) do
    begin
     $t \leftarrow 0$
     initiate $P(t)$
     evaluate $F(t)$
     while ($t < P$) do
       begin
         $t \leftarrow t + 1$
         select $P(t)$ from $P(t - 1)$
         crossover $P(t)$ according to crossover rate
         mutate $P(t)$ according to mutation rate
         evaluate $F(t)$
       end
     $i \leftarrow i + 1$
    end
End
```

Fig. 1. Genetic Algorithms Used in the Study

are selected to form new solutions (offspring) according to their fitness; the more suitable they are, the more chance they have to reproduce. This process is repeated over a series of 'generations' until some termination condition is satisfied. In the GA, the primary operations and parameters are as follows:

Selection Selection determines the chromosomes that are selected from the population to be parents for crossover, based on their fitness. There are many methods for selecting the best chromosomes such as roulette wheel, Boltzmann, tournament, rank and steady state [78]. The experiments reported upon in this paper use the elitism and rank selection method.

Crossover and Crossover Probability Crossover operates on selected genes (elements of chromosomes) from parent chromosomes to create new offspring. The likelihood that crossover will be performed is called Crossover Probability [78]. The experiments reported upon in this paper use the method of multi-point crossover with a Crossover Probability of 0.8.

Mutation and Mutation Probability Mutation randomly changes the offspring resulting from crossover. The likelihood of mutation is called the Mutation Probability [78]. The experiments reported upon in this paper use random bit flip with a Mutation Probability of 0.01.

```
Hill Climbing Algorithm:
Parameters: Max: $Population * Generation$ (referring to parameters of
GA); Sum: the calculation times of the fitness;
S: the current solution; N: the neighbour of the current solution.
Begin
  $Sum \leftarrow 0$
  while($Sum <= Max$) do
    begin
      initiate $S$ randomly
      if($S < HC(S)$)
          $S \leftarrow HC(S)$
    end
End

$HC(S)$
  $i \leftarrow 0$
  while($i < the\ number\ of\ all\ the\ slices$) do
    begin
      look for $N(i)$
        while(true) do
          begin
            $Sum \leftarrow Sum + 1$
            if($fitness\ of\ S < fitness\ of\ N(i)$)
                $S \leftarrow N(i)$
                $i \leftarrow 0$
                break
            else look for next neighbour of the current solution.
          end
        $i \leftarrow i + 1$
      end
    return current S
```

Fig. 2. Hill Climbing Algorithms Used in the Study

3.2 Hill Climbing

A Hill-Climbing (HC) algorithm looks for the neighbour of current solution and if the neighbor is better, this neighbour replaces the current solution. The operation will be repeated until no better neighbour can be found. In order to ensure fairness of comparison, the HC algorithm has the same budget of computation time. That is, the experiments use multiple restart Hill-Climbing and allow the same number of fitness evaluations in total (over all hill climbs) as are allowed to other algorithms.

```
Greedy Algorithm:
Parameters: Initial Solution Set: {0,0,0,...,0,0}; Candidate Set: the set of
all the slices of the program.
Begin
   evaluate each slice of candidate set
   while(not solution)
     begin
       select the slice
     end
End
```

Fig. 3. Greedy Algorithm Algorithms Used in the Study

```
Random Algorithm:
Parameters: Generation(G): 100; Individuals(I) (corresponding to the
population in GA): 50.
Begin
   $i \leftarrow 0$
   while($i < G$) do
      begin
         $t \leftarrow 0$
         while($t < I$) do
            begin
               initiate $I(t)$ randomly
               evaluate $F(t)$
               $t \leftarrow t + 1$
            end
         $i \leftarrow i + 1$
      end
End
```

Fig. 4. Random Algorithms Used in the Study

3.3 Greedy Algorithm

In general, a greedy algorithm consists of two sets and three main functions [63]:

Solution Set From which a solution is created.
Candidate Set Which represents all the possible elements that might compose the solution.
Selection Function Which chooses the most promising candidate to be added to the solution.
Value-Computing Function Which gives the value of a solution.
Solution Function Which checks whether a final solution has been reached.

In the experiments, the initial solution set is a binary string with each bit set to 0 and all the slices make up the candidate set; the value-computing function evaluates the number of program points of current solution set; the selection function chooses the slice that has the best contribution to the coverage value of solution and the smallest overlap value, that is, the bigger the ratio of increment of coverage and increment of overlap, the more chance the slice is choosen; the solution function checks whether coverage value of current solution has covered the whole program points in the program. The greedy algorithm is a heuristic algorithm and not a search algorithm, but its results can be compared to the others using the same fitness function.

3.4 Random Algorithm

The Random Algorithm generates the individuals (solutions) randomly. The purpose of using the Random Algorithm is to measure the performance of the other algorithms. Since a random search is unguided and therefore "unintelligent", it would be hoped that the guided search approaches and the greedy algorithm would outperform it. The random algorithm is therefore included to provide a base line, below which performance of the other algorithms should not fall.

4 Empirical Study

An empirical study was conducted to investigate the first three research questions described in Section 1. The slicing data used in the empirical study was collected by constructing the set of a possible backward slices (with $CodeSurfer$) of each program's System Dependence Graph (SDG) [44]. Slice size is measured by counting vertices of the dependence graph, rather than lines of code. The study concerns source codes of six open source programs, written in C. The program sizes range is from 37 to 1,008 program points. On the first sight, this may seem relatively small. However, the problem complexity is determined by the number of sets of slices which ranges from 2^{37} to 2^{1008} which is a very large search space. Summary of information concerning the programs studied can be formed in Table 2.

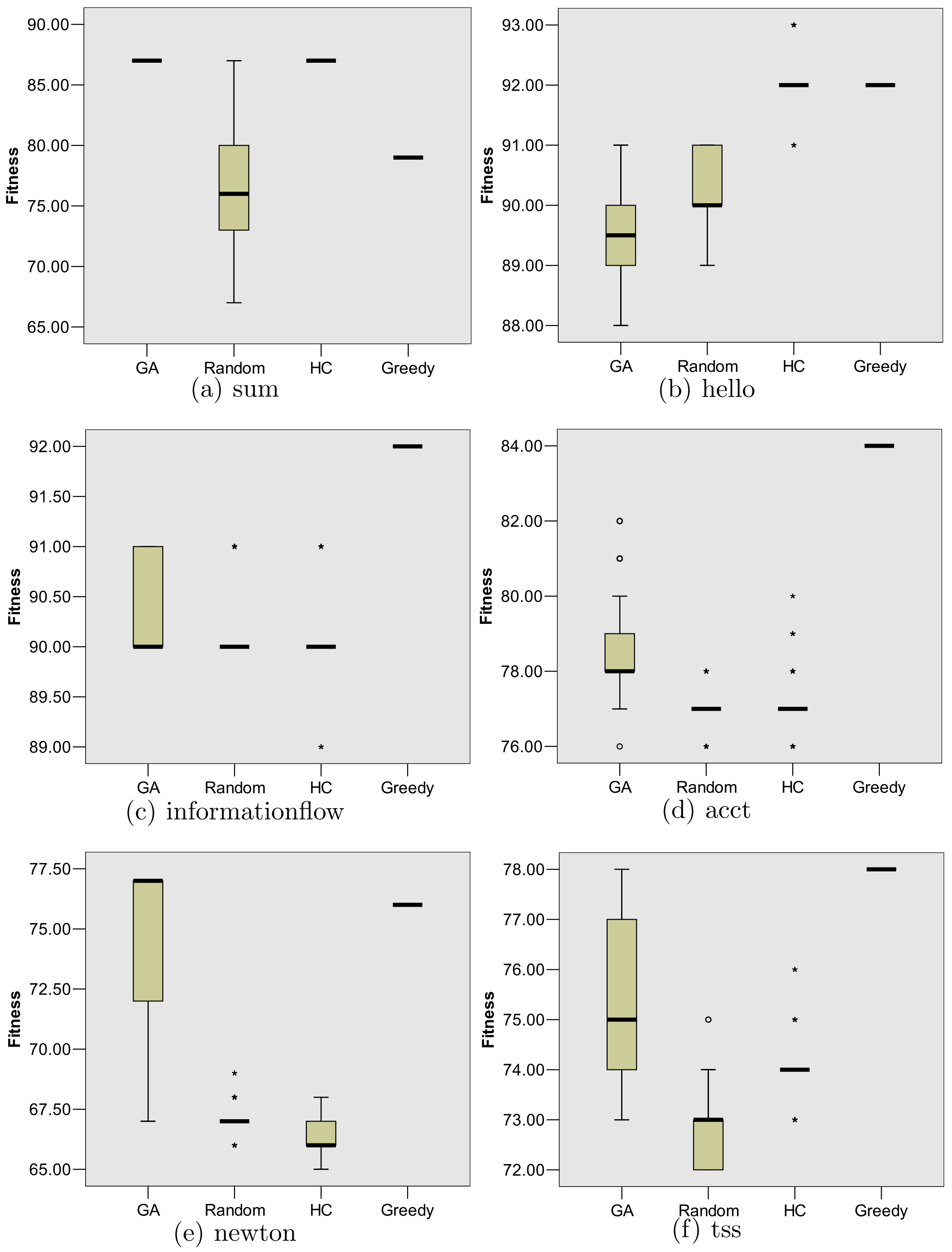


Fig. 5. Box plot of results for backward slicing in term of Fitness Function 1 defined in Section 2.3. The results show that the greedy algorithm performs the best. The low variance for the Hill Climbing algorithm (HC) suggests either a low order of modality in the landscape or a multi-modal landscape with similar valued peaks.

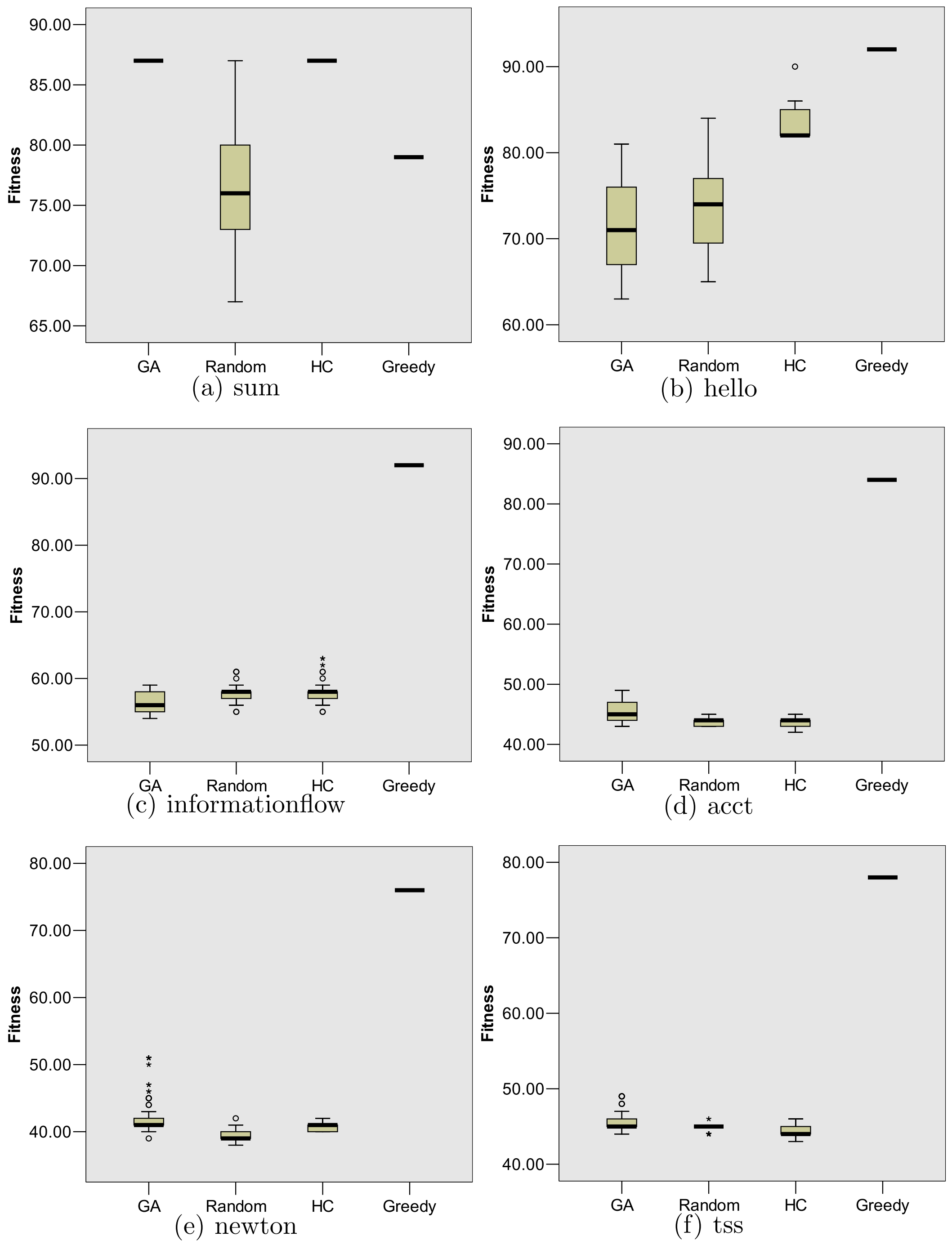


Fig. 6. Box plot results for backward slicing in term of Fitness Function 2 defined in section 2.3. The results confirm the result from Fitness Function 1 (presented in Figure 5) that the greedy algorithm performs the best. The programs are presented in increasing order of size (top-to-bottom, left-to-right), providing evidence that the gap in performance between the greedy algorithm and the others increases with program size. The low variance for the Hill Climbing Algorithm (HC) also replicates the finding for Fitness Function 1.

Programs	LoC	Program Point	Size of Search Space	Description
`Sum`	20	34	1.37×10^{11}	Numerical value calculation
`Hello`	43	76	7.55×10^{22}	Simple program, but more complex than 'hello world'
`Inform-`ationflow	109	176	9.57×10^{52}	Example of simple information processing
`Acct`	681	546	2.30×10^{164}	Accounting package
`Newton`	819	998	2.67×10^{300}	Interpolated polynomial that uses Newton's method
`Tss`	896	1008	2.74×10^{303}	Three kinds of mathematical interpolation function
Total	2,568	2,838	2.74×10^{303}	

Table 2
The subject programs studied.

4.1 Which Algorithm Is the Best?

Each non-greedy algorithm was executed 100 times with randomly chosen initial values (thus effectively sampling the space of possible start points). This produces a set of 100 results, one for each execution. The results obtained for some particular execution is determined by the random seed. The population from which this sample of 100 execution comes, is thus the population of random seeds. For the Greedy Algorithm, the execution results are the same every time since the results are gained with 'Greedy Strategy', rather than the random initial population.

Under the first fitness function (results presented in Figure 5), the performance of the Greedy algorithm is the best except for the smallest program *sum*. Moreover, it is observed that for the smaller programs (e.g. (a) and (b)) HC performs better than either GA or Random. As program size increases (e.g. (c)-(f)) the GA performs better, beating HC and Random. HC performs worse on the larger programs suggesting that the HC landscape is too flat to easily find maxima.

To determine the relative performance of these three algorithms (non greedy algorithms), the Mann-Whitney and Wilcoxon test is applied to every program. As mentioned above, 100 sample results from execution are gathered for each program. For each program, the difference between the set of samples for each algorithm is significant due to Asymp. Sig. (2-tailed)=.000 and Exact Sig.(2-tailed)=.000) at $p = .05$.

Under the second fitness function (results presented in Figure 6), similar characteristics can be observed to the first fitness function. The Greedy algorithm outperforms the others except for the smallest program *sum*. The GA performs the best of the non-greedy algorithms and the HC algorithm does not improve on Random except in the smallest program (a). In the same way, the Mann-Whitney and Wilcoxon test at $p = .05$ applied in each program finds that Asymp. Sig. (2-tailed)=.000 and Exact Sig.(2-tailed)=.000 represent that the difference of values of GA, Random and HC is statistically significant in each program.

In summary, the GA performs better as program size increases with the HC algorithm having the opposite characteristic. Random is beaten by GA in all programs but by the HC algorithm only in small programs. The Greedy algorithm beats the other 3 algorithms in the most situations. Furthermore, Table 3 shows the execution time of each algorithm for each program, which suggests that the Greedy algorithm also has the best performance among 4 algorithms.

Programs	Greedy		GA		HC		Random	
	F1	F2	F1	F2	F1	F2	F1	F2
Sum	11	11	15121	15640	14902	10985	14525	14859
Hello	15	15	15611	16471	17760	15484	13250	16096
Information-	20	20	16611	17019	15531	14953	16353	16768
Acct	46	46	30676	42007	31553	21937	27256	36534
Newton	139	139	109122	166521	112229	97794	99665	161412
Tss	140	140	119124	183091	127693	103511	111678	171653

Table 3
Execution time of each algorithm for each program in condition of the machine—Ram 512M; Pentium4 3.2GHz. F1 and F2 represent the fitness function (1) and (2) defined in section 2.3, respectively; the measurement is based upon the milli second.

4.2 How similar are the results for each algorithm?

This section presents two approaches to compare the results produced by each algorithm for similarity. The first is a purely visual representation, used to provide visual evidence for similarity. The second is a quantitative assessment of the similarity of results. The findings suggest that the algorithms find similar (though not identical) solutions. This level of agreement, coupled with the low variance in hill climbing results provides evidence that the landscape is either uni-modal or multi-modal but with many local optima of similar value to each other, with the result that it is possible for search algorithms to find solutions of reasonable quality.

4.2.1 Qualitative Similarity Analysis

Figures 7 to 12 provide a visualization of the results of the search. The goal of visualizing the optimal slicing set is to display the result obtained. Thus, the figures show optimal slicing set with search algorithms, rather than the entire set of slices for all the program points.

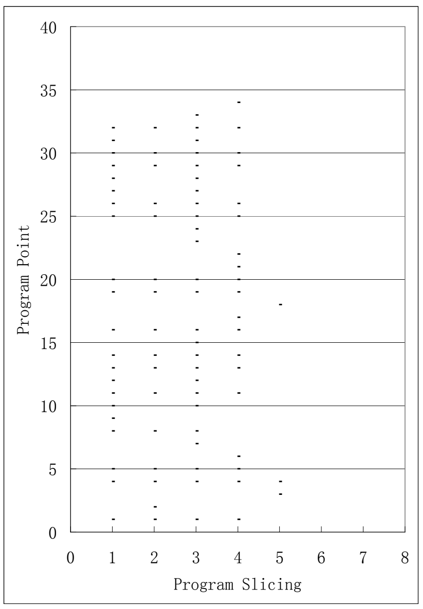
The X axis represents the slicing criteria, ordered by their relative location in the source code: earlier source code locations appear closer to the left side. The Y axis represents the program points belonging to the corresponding slice. As can be seen from these figures, the results produced by each algorithm are strikingly similar (Especially for the *newton* and *tss*, the image appears almost the same) but not identical.

In the Figures 11 and 12, it can be seen that there is a greater degree of similarity in the three heuristic methods (greedy, GA and HC), while the Random algorithm appears to produce rather less ‘coherent image’. On the other hand, the greedy algorithm is apt to find the optimal slicing set which has less slices than the GA, HC and random algorithm (except for the *sum* due to so small program points in the program–too tiny). That is, the greedy can find optimum solutions which have smallest slices in the slicing set, such that the decomposition of program has the simplest form.

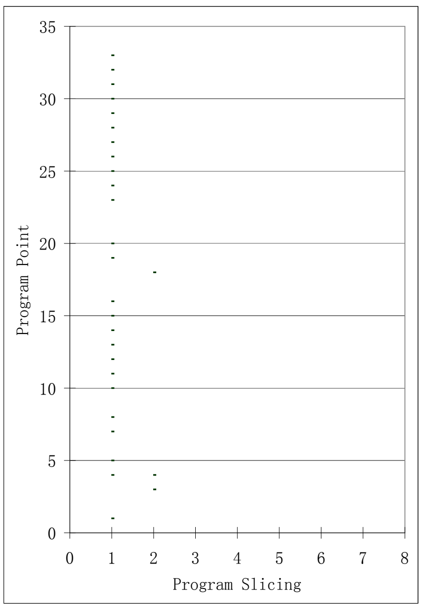
Moreover, GA and HC perform better than random algorithm, which can be observed from Figures 5 and 6(also from the empirical study in section 4.1), since GA and HC always try to cover with the program points in the program as many as possible, whereas the random algorithm has the form of less coverage–less coherent image. However, GA sometimes finds the slicing set which has more overlap than three others. Of course, these observations are qualitative and of illustrative value only. The next subsection provides a quantitative similarity analysis.

4.2.2 Quantitative Similarity Analysis

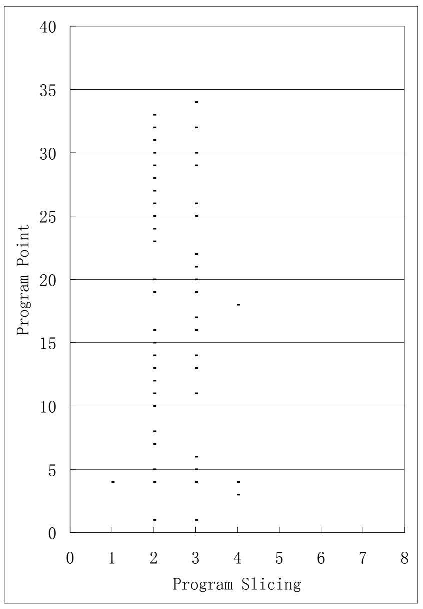
Table 4 presents results concerning the quantity of agreement between each pair of results for each algorithm. The calculation used for this is: $100 \cdot \frac{\cap(A,B)}{Min(A,B)}$. This represents, as a percentage, the degree of agreement between two sets A and B. If the sets are identical then agreement is 100%; if there is no intersection then agreement is 0%. The percentage agreement measures the size of the intersection between the two sets relative to the size of the smaller of the two. Therefore, it is a measure of the degree to which the maximum possible intersection size is achieved.



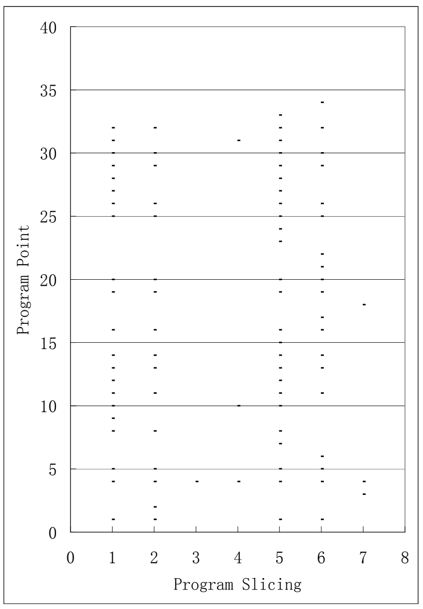
(a) Greedy (Fitness Value = 88)



(b) GA (Fitness Value = 88)

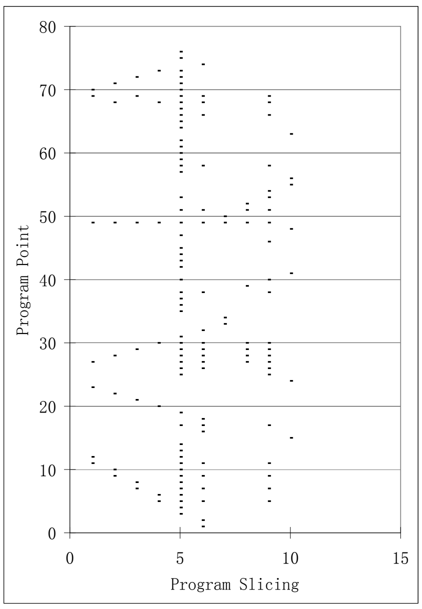


(c) HC (Fitness Value = 88)

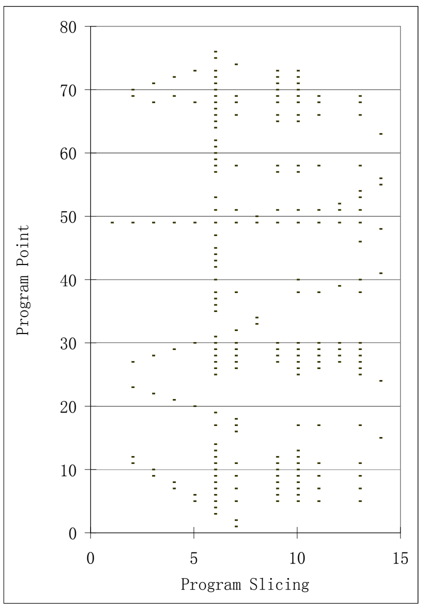


(d) Random (Fitness Value = 87)

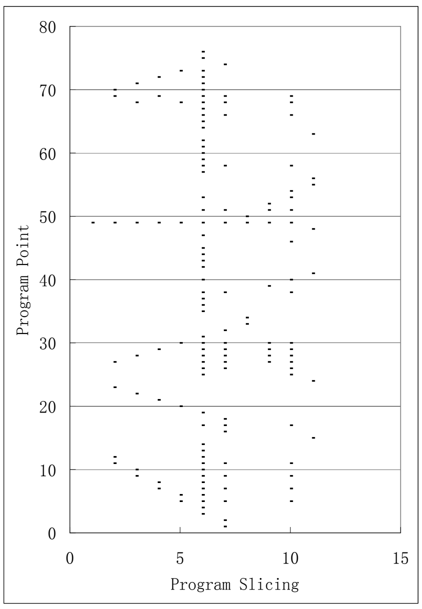
Fig. 7. Visualized results for backward slicing based on Fitness Function 1 with the program *sum*. The GA produces smallest results for this very small program.



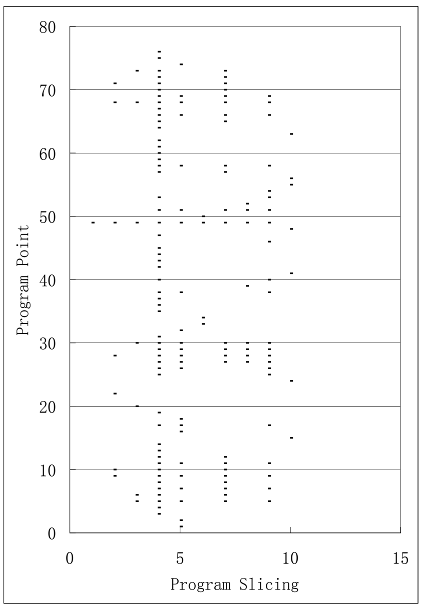
(a) Greedy (Fitness Value = 92)



(b) GA(Fitness Value = 90)

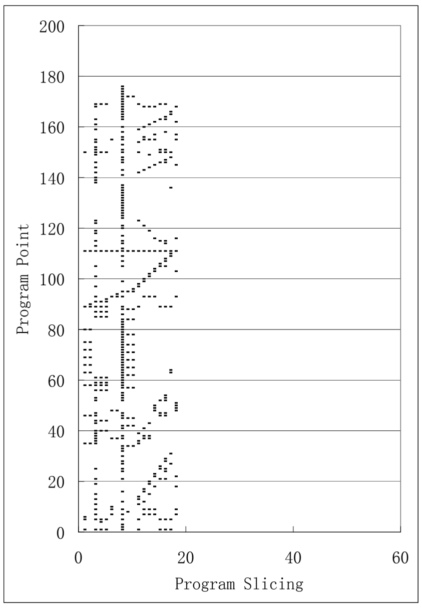


(c) HC (Fitness Value = 92)

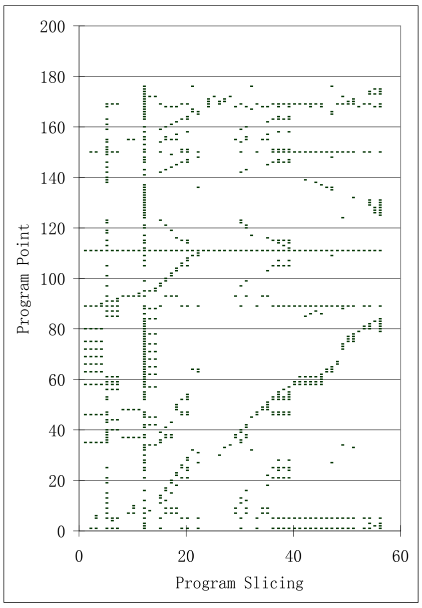


(d) Random (Fitness Value = 91)

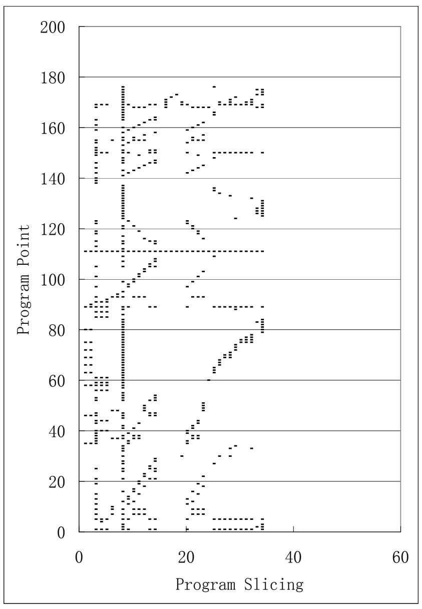
Fig. 8. Visualized results for backward slicing based on Fitness Function 1 with the program *hello*.



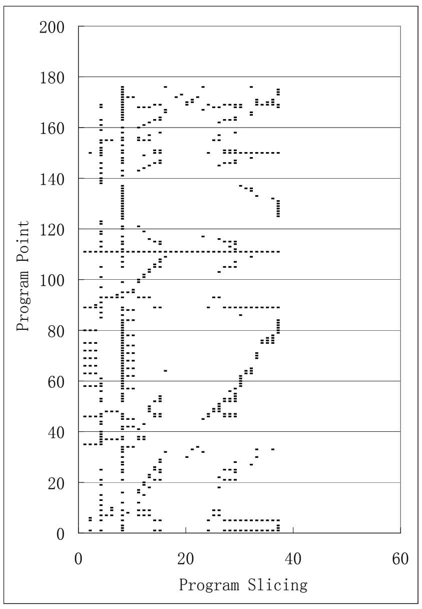
(a) Greedy (Fitness Value = 92)



(b) GA (Fitness Value = 90)

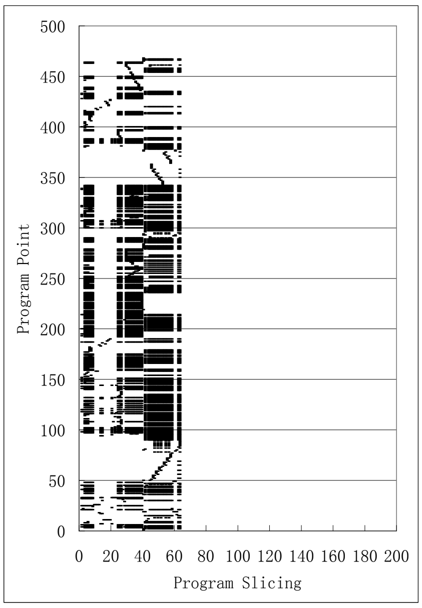


(c) HC (Fitness Value = 90)

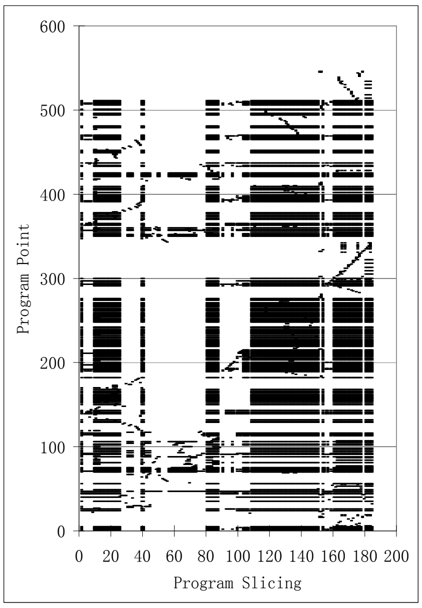


(d) Random (Fitness Value = 90)

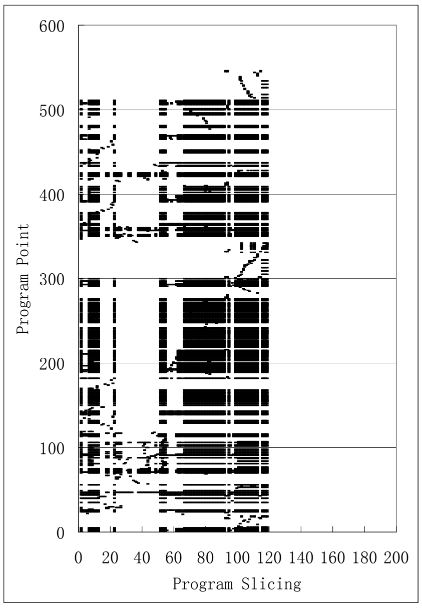
Fig. 9. Visualized results for backward slicing based on Fitness Function 1 with the program *informationflow*. Note that the greedy algorithm produces the best results and also achieves this with the fewest slices.



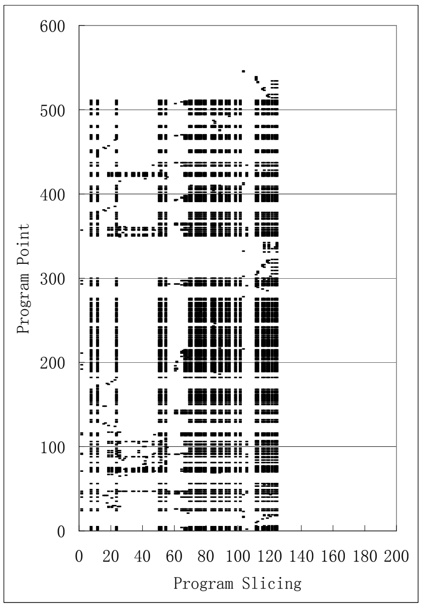
(a) Greedy (Fitness Value = 84)



(b) GA (Fitness Value = 80)

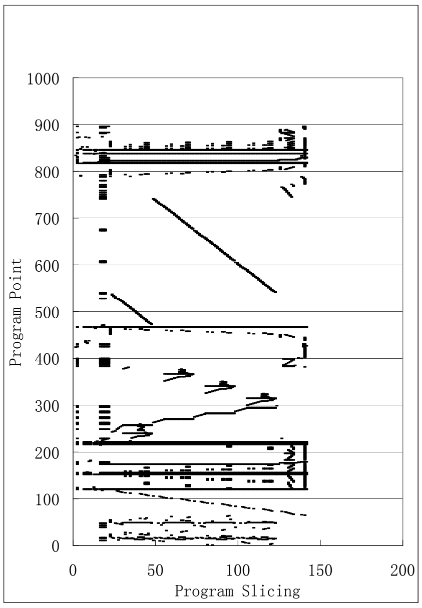


(c) HC (Fitness Value = 78)

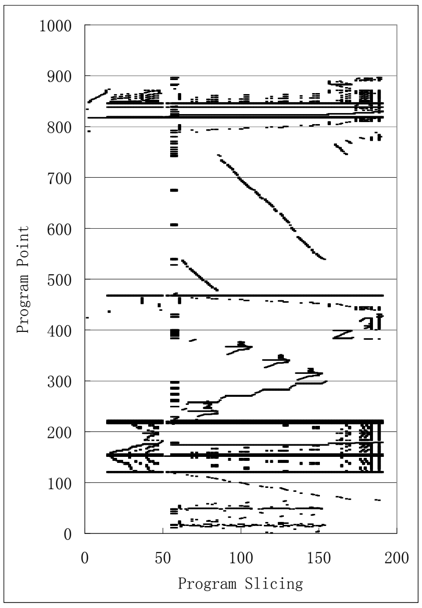


(d) Random(Fitness Value = 77)

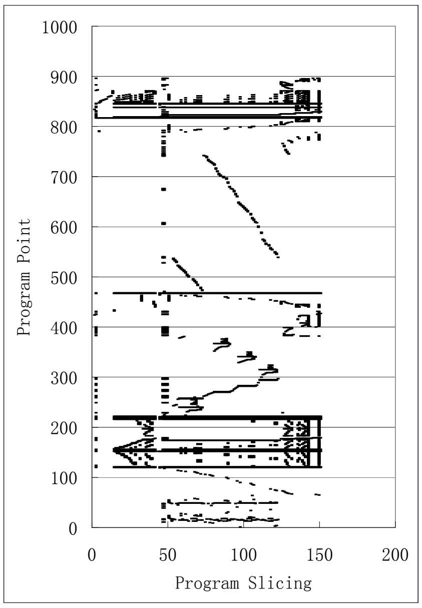
Fig. 10. Visualized results for backward slicing based on Fitness Function 1 with the program *acct*. The greedy algorithm produces the best results with the fewest slices.



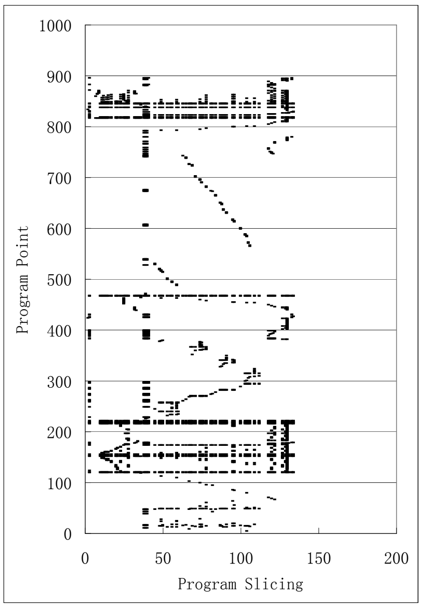
(a) Greedy (Fitness Value = 77)



(b) GA (Fitness Value = 76)

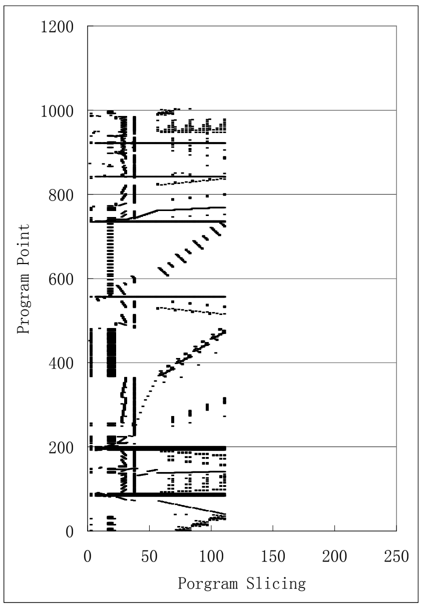


(c) HC (Fitness Value = 68)

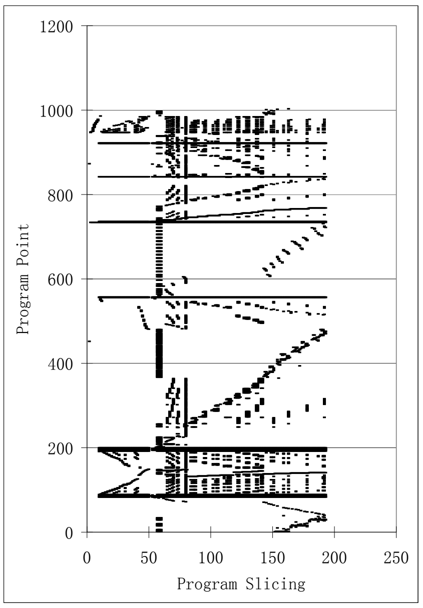


(d) Random (Fitness Value = 66)

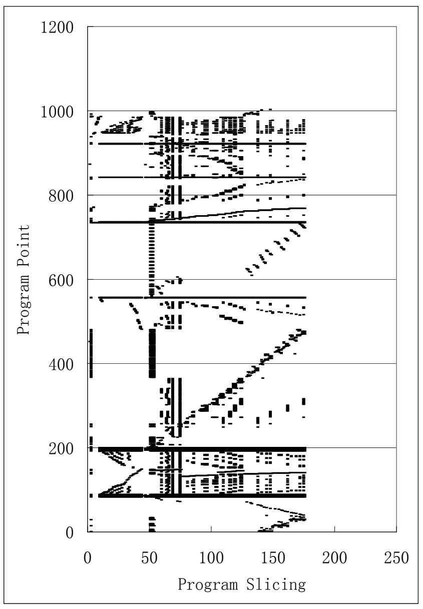
Fig. 11. Visualized results for backward slicing based on Fitness Function 1 with the program *newton*. Note the image for the Random search appears to be a 'grainy' version of that for the others and that the greedy algorithm result contains fewer slices. There are more similarity in the intelligent searches; random produces a 'poor imitation'.



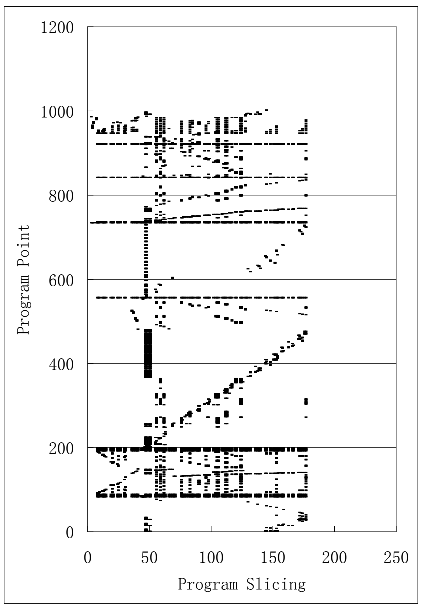
(a) Greedy (Fitness Value = 88)



(b) GA (Fitness Value = 50)



(c) HC (Fitness Value = 47)



(d) Random (Fitness Value = 43)

Fig. 12. Visualized results for backward slicing based on Fitness Function 1 with the program *tss*. Note the image for the Random search appears to be a 'grainy' version of that for the others and that the greedy algorithm result contains fewer slices.

By comparing how many identical slices there are in each pair of slicing sets, it is possible to measure result similarity. That is, the Table 4 indicates the level of agreement the different search techniques share as to their choices of optimal solution.

4.3 Visual Evidence for the Presence for Clones

Clone detection is referred to as techniques to detect duplicated code in programs. Clone detection techniques have been widely investigated and can be roughly classified into three categories: string-based [28,45], token-based [6,46], parse-tree based [9,50,58], which have different performance with refactoring tools to remove duplicated code [70]. Moreover, Komondoor [49] and Krinke [51] use dependence to identify clone code.

The visualization of results yields an unexpected but interesting finding related to the presence of clones. Notice the repeated patterns in Figures 13, 14, 15 and 16. There are two kinds of repeated patterns. The first kind are examples of sharing the same program points. In these patterns, the same vertical image is replicated across the X axis, for example, the middle section of Figure 10 (b) where the number of program points is between 200 and 300. This is an example of a situation where a whole series of slices share the same subset of nodes in their slices.

However, there are also some potentially more interesting repeated images. Those are not dependence clusters, because they do not share a set of y axis points. For example, consider the four blocks in Figures 14 and 16 (A, B, C and D). These images denote patterns of dependence that are repeated in different sections of the code. For instance, if one scans the program *newton*, the code related to similar blocks in Figure 14 is shown in the Figure 13. The corresponding four blocks of the codes compute the four interpolated coefficients with the different inputs. For the code of *tss* as shown in Figure 15, the corresponding blocks A, B, C and D in Figure 16 represent 4 functions which compute three kinds of mathematical interpolation in different parameters, respectively. In each group of 4 blocks of codes, a similar functionality emerges.

Inspection of the code quickly reveals that the four chunks of code are clones. However, they are not identical. Nonetheless, they have a similar dependence structure which shows up in the visualisation. Because the search seeks to cover the program, these similar figures occur at different program points they tend to show up.

When these two groups of duplicate code are mapped to the visualization of slicing sets shown in Figures 14 and 16 respectively, the corresponding blocks

(a) Program1 sum.c

Algorithms	Greedy	Genetic	H-Climbing	Random
Greedy	N/A	40	60	71
Genetic	40	N/A	50	29
H-Climbing	60	50	N/A	57
Random	71	29	57	N/A

(b) Program2 hello.c

Algorithms	Greedy	Genetic	H-Climbing	Random
Greedy	N/A	71	91	80
Genetic	71	N/A	79	71
H-Climbing	91	79	N/A	82
Random	80	71	82	N/A

(c) Program3 informationflow.c

Algorithms	Greedy	Genetic	H-Climbing	Random
Greedy	N/A	32	41	38
Genetic	32	N/A	48	61
H-Climbing	41	48	N/A	59
Random	38	61	59	N/A

(d) Program4 acct.c

Algorithms	Greedy	Genetic	H-Climbing	Random
Greedy	N/A	28	34	32
Genetic	28	N/A	53	55
H-Climbing	34	53	N/A	56
Random	32	55	56	N/A

(e) Program5 newton.c

Algorithms	Greedy	Genetic	H-Climbing	Random
Greedy	N/A	54	56	57
Genetic	54	N/A	58	61
H-Climbing	56	58	N/A	59
Random	57	61	59	N/A

(f) Program6 tss.c

Algorithms	Greedy	Genetic	H-Climbing	Random
Greedy	N/A	38	36	32
Genetic	38	N/A	49	57
H-Climbing	36	49	N/A	51
Random	32	57	51	N/A

Table 4
Comparison of slicing sets between the programs of the source code, $100 \cdot \frac{\cap(A,B)}{Min(A,B)}$.

are denoted by a similar shape, which suggests the presence of clones. Consider the program *Newton* as an illustrated example. The program computes the outputs of four interpolated coefficients with the different inputs, and the computation of each coefficient is dependent on a corresponding block of code. The code for each of the four is very similar. The information can be captured with visualization of slices of the set of some program points contributing to computation of the coefficients.

This is interesting and may suggest applications for search based slicing in clone detection. However, this remains a topic for future work, as clone detection is not the focus of the present paper.

4.4 Flexibility of the Framework

This paper introduces the general framework that applying the Search Based Software Engineering theory to Program Slicing looks for the interesting Dependence Structures in the source code. The authors define the fitness function that can decompose the program into a slicing set in which the overlaps might be minimum. This is only an illustrated example to demonstrate the possible application of the framework. In fact, search based slicing could be some other potential applications in source code analyses. The following section will introduce three feasible applications with this framework and researchers might define different fitness functions according to the specific purposes for different problems.

4.4.1 Splitting/Refactroing functions/procedures to improve cohesion

In general, a function (or procedure) in a program independently computes one or multi results and return outputs by defining some processing elements [68]. Functions have different cohesion levels, which determine the readability, testability, and maintainability of software in terms of the relationship between these processing elements. High cohesion is usually considered to be desirable [14,2]. According to the definition of Ott and Thuss [68], cohesion levels can be divided into 4 classifications: low, control, data and function as depicted in Figure 17.

The goal of splitting a function is to reconstruct the original function which has the lower cohesion into the set of subfunctions which all have higher cohesion, without changing original semantics. The hope is that each smaller function is more reusable and robust.

The low level suggests several distinct unrelated processing elements; the control level case is similar to the low level except that processing elements is all

```
log_file("***newtoncoefficient\n number=6：**********\n");
    strcat(str_tmp1,"\n independent variable x：");
    strcat(str_tmp2,"\n f(x)：");
    strcat(str_tmp3,"\n newton N(x)：");
    for(i = 0;i <= 5 ; i ++)
    {
        memset(&str_vtmp,0,sizeof(str_vtmp));
        sprintf(str_vtmp,"%f ",x1[i]);
        strcat(str_tmp1,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        sprintf(str_vtmp,"%f ",y1[i]);
        strcat(str_tmp2,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        memset(&str_tt,0,sizeof(str_tt));
        memset(&str_new,0,8192);
        if(i != 0){
        for(j = i    ; j > 0 ; j --)
        {
            memset(&str_tt,0,sizeof(str_tt));
            if(x1[j-1] <0)
                sprintf(str_tt,"*(x + %f)",((-1.0)*x1[j-1]));
            else sprintf(str_tt,"*(x - %f)",x1[j-1]);
            strcat(str_new,str_tt);
        }
        }
        if (i == 0) sprintf(str_vtmp,"%f ",n1[i]);
        else {
            if(n1[i]>0) sprintf(str_vtmp," + %f " , n1[i]);
            else sprintf(str_vtmp," %f " , n1[i]);
        }
        strcat(str_vtmp,str_new);
        strcat(str_tmp3,str_vtmp);
    }
```

(a) Block A

```
log_file("\n*****the number=11：*****\n\n");
    memset(&str_tmp1,0,sizeof(str_tmp1));
    memset(&str_tmp2,0,sizeof(str_tmp2));
    memset(&str_tmp3,0,sizeof(str_tmp3));
    strcat(str_tmp1,"    independent variable x：");
    strcat(str_tmp2,"\n f(x)：");
    strcat(str_tmp3,"\nnewton N(x)：");
    for(i = 0;i <= 10 ; i ++)
    {
        memset(&str_vtmp,0,sizeof(str_vtmp));
        sprintf(str_vtmp,"%f ",x2[i]);
        strcat(str_tmp1,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        sprintf(str_vtmp,"%f ",y2[i]);
        strcat(str_tmp2,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        memset(&str_tt,0,sizeof(str_tt));
        memset(&str_new,0,8192);
        if(i != 0){
        for(j = i    ; j > 0 ; j --)
        {
            memset(&str_tt,0,sizeof(str_tt));
            if(x2[j-1] <0)
                sprintf(str_tt,"*(x + %f)",((-1.0)*x2[j-1]));
            else sprintf(str_tt,"*(x - %f)",x2[j-1]);
            strcat(str_new,str_tt);
        }
        }
        if (i == 0) sprintf(str_vtmp,"%f ",n2[i]);
        else {
            if(n2[i]>0) sprintf(str_vtmp," +
            else sprintf(str_vtmp," %f " ,
        }
        strcat(str_vtmp,str_new);
        strcat(str_tmp3,str_vtmp);
    }
```

(b) Block B

```
log_file("\n********the number=16：******\n\n");
    memset(&str_tmp1,0,sizeof(str_tmp1));
    memset(&str_tmp2,0,sizeof(str_tmp2));
    memset(&str_tmp3,0,sizeof(str_tmp3));
    strcat(str_tmp1," independent variable x：");
    strcat(str_tmp2,"\n f(x)的：");
    strcat(str_tmp3,"\nnewton N(x)：");
    for(i = 0;i <= 15 ; i ++)
    {
        memset(&str_vtmp,0,sizeof(str_vtmp));
        sprintf(str_vtmp,"%f ",x3[i]);
        strcat(str_tmp1,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        sprintf(str_vtmp,"%f ",y3[i]);
        strcat(str_tmp2,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        memset(&str_tt,0,sizeof(str_tt));
        memset(&str_new,0,8192);
        if(i != 0){
        for(j = i    ; j > 0 ; j --)
        {
            memset(&str_tt,0,sizeof(str_tt));
            if(x3[j-1] <0)
                sprintf(str_tt,"*(x + %f)",((-1.0)*x3[j-1]));
            else sprintf(str_tt,"*(x - %f)",x3[j-1]);
            strcat(str_new,str_tt);
        }
        }
        if (i == 0) sprintf(str_vtmp,"%f ",n3[i]);
        else {
            if(n3[i]>0) sprintf(str_v
                        else sprintf(str_vtmp," %f " , n3[i]);
        }
        strcat(str_vtmp,str_new);
        strcat(str_tmp3,str_vtmp);
    }
```

(c) Block C

```
log_file("\n*********the number=21：*************\n\n");
    memset(&str_tmp1,0,sizeof(str_tmp1));
    memset(&str_tmp2,0,sizeof(str_tmp2));
    memset(&str_tmp3,0,sizeof(str_tmp3));
    strcat(str_tmp1," independent variable x：");
    strcat(str_tmp2,"\n f(x)：");
    strcat(str_tmp3,"\nnewton N(x)：");
    for(i = 0;i <= 20 ; i ++)
    {memset(&str_vtmp,0,sizeof(str_vtmp));
    sprintf(str_vtmp,"%f ",x4[i]);
    strcat(str_tmp1,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
    sprintf(str_vtmp,"%f ",y4[i]);
    strcat(str_tmp2,str_vtmp);
        memset(&str_vtmp,0,sizeof(str_vtmp));
        memset(&str_tt,0,sizeof(str_tt));
        memset(&str_new,0,8192);
    if(i != 0){
    for(j = i    ; j > 0 ; j --)
    {memset(&str_tt,0,sizeof(str_tt));
      if(x4[j-1] <0)
      sprintf(str_tt,"*(x +
          %f)",((-1.0)*x4[j-1]));
      else sprintf(str_tt,"*(x -%f)",x4[j-1]);
      strcat(str_new,str_tt);
      if (i == 0)
        sprintf(str_vtmp,"%f ",n4[i]);
      else {if(n4[i]>0)
          else sprintf(str
        }
        strcat(str_vtmp,str_new);
        strcat(str_tmp3,str_vtmp);
    }
```

(d) Block D

Fig. 13. clones present in the program *newton*

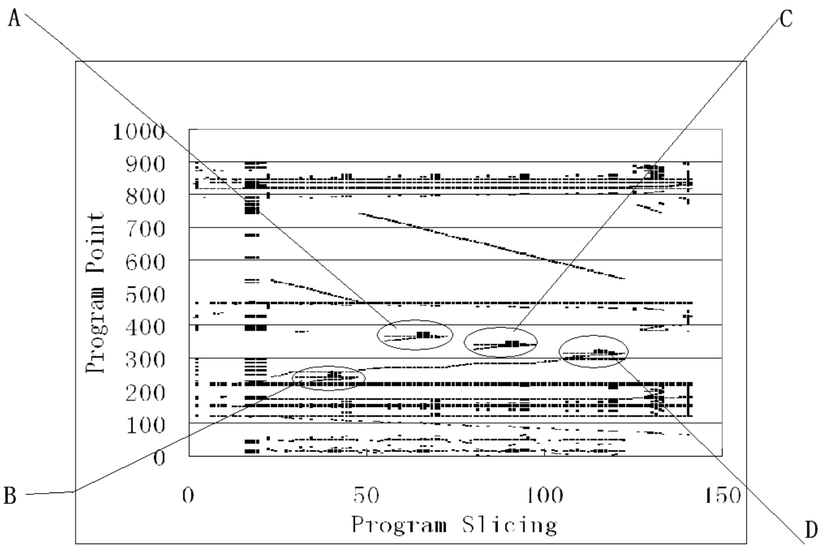


Fig. 14. Clone detection to *newton*

```
void    TSS1(float arr_low1[5],float arr_up1[5],float d1[6],float arr_res1[6])
{           int k;
            int arr_diag[6] = {2,2,2,2,2,2};
            float arr_q[6] ;
            float arr_y[6];
            float arr_p[5];
            arr_q[0]= arr_diag[0];
            arr_y[0]= d1[0];
            for(k = 2;k <= 6;k ++) {
            arr_p[k-2]= arr_low1[k-2]/arr_q[k-2];
            arr_q[k-1]= arr_diag[k-1]-arr_p[k-2]*arr_up1[k-2];
            arr_y[k-1]= d1[k-1]-arr_p[k-2]*arr_y[k-2]; }
            arr_res1[5]= arr_y[5]/arr_q[5];
            for(k = 6 - 2 ;k >= 0;k --) {
            arr_res1[k]=(arr_y[k]-arr_up1[k]*arr_res1[k+1])/arr_q[k];}
            return;}
```

(a) Block A

```
void    TSS2(float arr_low2[10],float arr_up2[10],float d2[10],float arr_res2[10])
{           int k;
            int arr_diag[11] = {2,2,2,2,2,2,2,2,2,2,2};
            float arr_q[11] ;
            float arr_y[11];
            float arr_p[10];
            arr_q[0]= arr_diag[0];
            arr_y[0]= d2[0];
            for(k = 2;k <= 11;k ++) {
            arr_p[k-2]= arr_low2[k-2]/arr_q[k-2];
            arr_q[k-1]= arr_diag[k-1]-arr_p[k-2]*arr_up1[k-2];
            arr_y[k-1]= d1[k-1]-arr_p[k-2]*arr_y[k-2]; }
            arr_res2[5]= arr_y[10]/arr_q[10];
            for(k = 11 - 2 ;k >= 0;k --) {
            arr_res2[k]=(arr_y[k]-arr_up2[k]*arr_res2[k+1])/arr_q[k];}
            return;}
```

(b) Block B

```
void    TSS3(float arr_low3[15],float arr_up3[15],float d3[16],float arr_res3[16])
{         int k;
            int arr_diag[16] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
            float arr_q[16] ;
            float arr_y[16];
            float arr_p[15];
            arr_q[0]= arr_diag[0];
            arr_y[0]= d3[0];
            for(k = 2;k <= 16;k ++)       {
            arr_p[k-2]= arr_low3[k-2]/arr_q[k-2];
            arr_q[k-1]= arr_diag[k-1]-arr_p[k-2]*arr_up3[k-2];
            arr_y[k-1]= d3[k-1]-arr_p[k-2]*arr_y[k-2]; }
            arr_res3[15]= arr_y[15]/arr_q[15];
            for(k = 16 - 2 ;k >= 0;k --) {
            arr_res3[k]=(arr_y[k]-arr_up3[k]*arr_res3[k+1])/arr_q[k];}
            return;}
```

(c) Block C

```
void    TSS4(float arr_low4[20],float arr_up4[20],float d4[21],float arr_res4[21])
{         int k;
            int arr_diag[21] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
            float arr_q[21] ;
            float arr_y[21];
            float arr_p[20];
            arr_q[0]= arr_diag[0];
            arr_y[0]= d4[0];
            for(k = 2;k <= 21;k ++) {
            arr_p[k-2]= arr_low4[k-2]/arr_q[k-2];
            arr_q[k-1]= arr_diag[k-1]-arr_p[k-2]*arr_up4[k-2];
            arr_y[k-1]= d4[k-1]-arr_p[k-2]*arr_y[k-2];}
            arr_res4[20]= arr_y[20]/arr_q[20];
            for(k = 21 - 2 ;k >= 0;k --) {
            arr_res4[k]=(arr_y[k]-arr_up4[k]*arr_res4[k+1])/arr_q[k];}
            return;}
```

(d) Block D

Fig. 15. clone codes in *tss*

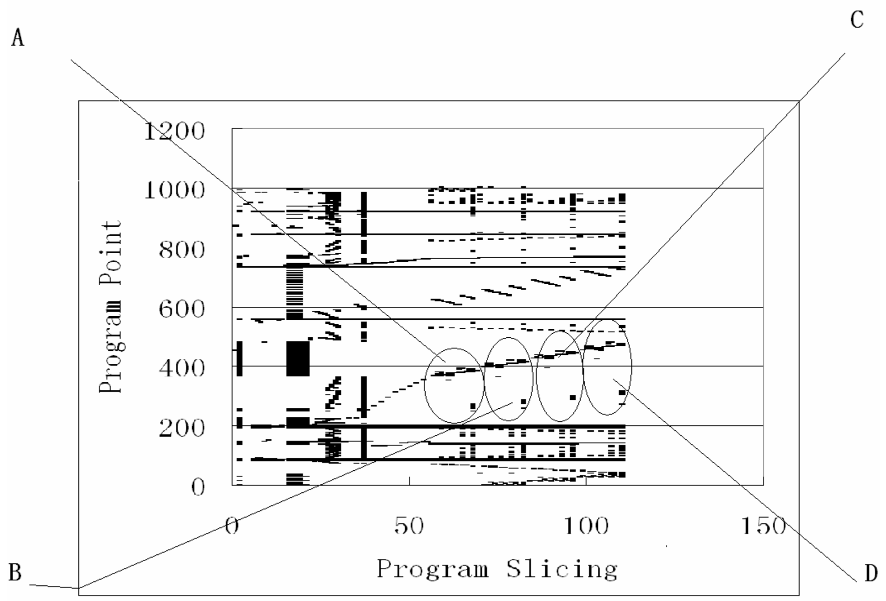


Fig. 16. Clone detection to *tss*

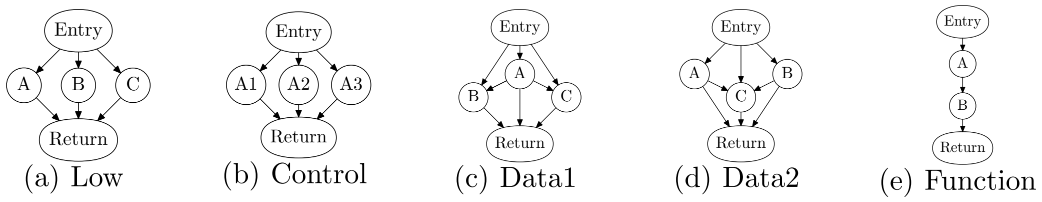


Fig. 17. Four classifications of function cohesion. The A, B and C represent the processing elements; A1, A2 and A3 represent the processing elements are all in the same control block such as *if*, *for* or *while*.

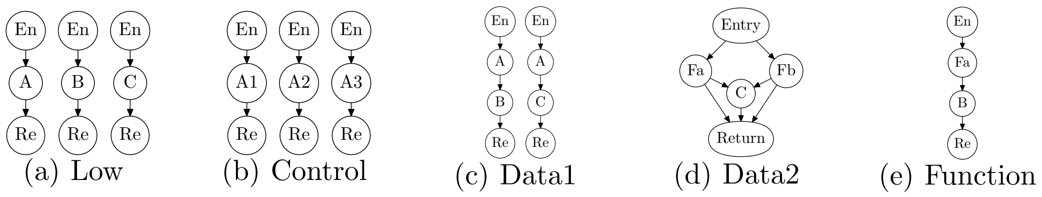


Fig. 18. The results of splitting or refactoring the functions. En and Re represent subfunction entry and return value, respectively; Fa and Fb represent function calling sites to subfunctions of processing element A and B, respectively. Note that the new subfunctions in (b) all includes control statements in which block A1, A2 and A3 belong.

dependent on some control statements. In this situation, search approaches can find several sets of slices, each of which represents a processing element, thereby in condition of no overlap or minimum overlap (or only control statements) allowing the function to be split out into subfunctions, hopefully with higher cohesion levels. The results of splitting the function are shown in Figure 18 (a) and (b). For the data level, there are two cases. Figure 17 (c) suggests that the computation of the processing elements B and C depend upon the results of the element A. This function also can be divided into two subfunctions by putting the element A into each subfunction computing B and C in the Figure 18 (c).

On the other hand, The Figure 17 (d) and (e) show 'non-split-function' cases. However, search approaches can look for several sets of slices corresponding to the processing elements A, B in the subfigure (d) and A in the subfigure (e). In this situation, a set/sets of slices representing the processing element A and/or B can be extracted from the functions into a new subfunction and position of the A and/or B can be replaced by function calling. The results of refactoring the function are shown in Figure 18 (d) and (e).

4.4.2 Parallelism Computation

Parallelizability can be measured as the number of slices which have a pairwise overlap less than a certain threshold. A high degree of parallelizability would suggest that assigning a processor to execute each slice in parallel could give a significant program speed-up [77].

Search approaches can seek to find specific combinations of slices which can reach such a threshold. Therefore, the fitness function can be described as:

> "Seek to search for a set of slices $Slice_1, ..., Slice_n$, in which the overlap of each pairwise is all less than the parameter of parallelism (to be defined in terms of the specific problem), such that the influences among the slices are least when each slice is given a separate processor."

4.4.3 'The Chain of Slices' for Program Comprehension

Normally, some slices in a program have overlaps, other than complete 'independence'. Especially, some slices could completely include the others or some slices are identical, for example, in the same dependence cluster [11]. Therefore, search approaches could look for several sets of slices, in each of which bigger slices can cover the smaller one, which is like a set of a chain of slices.

Therefore, the fitness function can be described as:

"Seek to search for a set of sets of slices $\{S_1, ..., S_n\}$ containing the whole program, in which $S_1, ..., S_n$ each represent a set of slices $Slice_1, ..., Slice_m$ which are chosen by the criteria: $Slice_i \subset Slice_{i+1}$. That is:
$\forall x \in Slice_i \Rightarrow x \in Slice_{i+1} \quad (0 < i < m)$"

This idea is to find out some inclusive relationship amongst slices such that it can contribute to comprehension for understanding the program. When maintainers want to catch on the program developed by other programmers, 'the chain' is helpful for comprehending the program structure step by step by understanding a program from the smallest slice to biggest one.

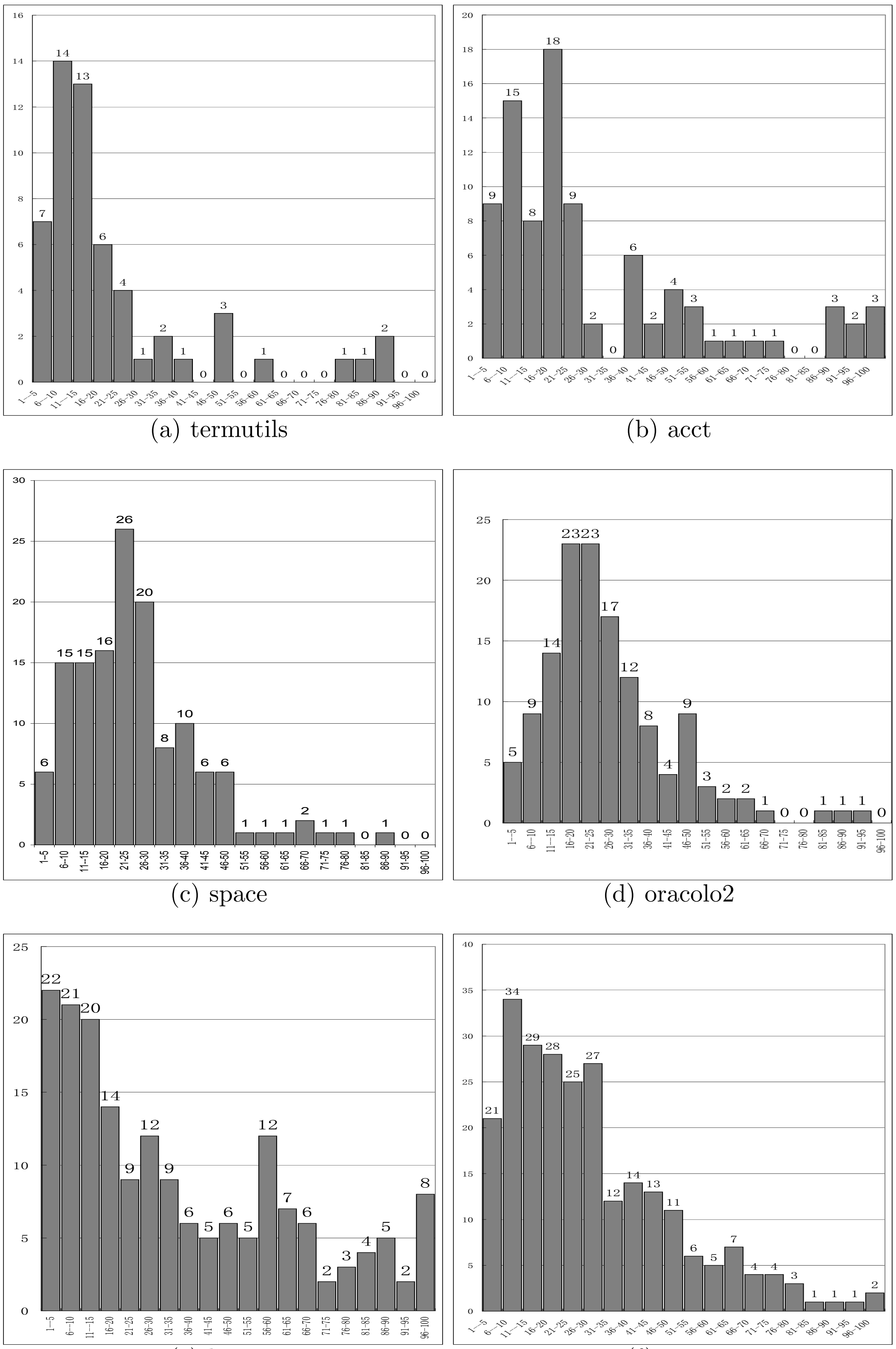
5 Further Empirical Study

In our previous experiments in Section 4, all 6 programs studied were relatively small since the purpose was to demonstrate that the SBSE framework can be applied to program slicing, to locate the interesting dependence structures in source code. At the same time, it was found that the Greedy algorithm was the best of the 4 algorithms in terms of its fitness function value, its execution time and the number of the chosen slices in the solution it finds.

Our further empirical study, reported below, applies the Greedy algorithm to 6 different larger C programs as shown in Table 5. These programs are all open source, drawn from the 'Gnu' website (ftp://ftp.gnu.org/gnu). The vertices are the program points in the SDG [43] and the slices are based on contributions from source code only. This is because, for the decomposition problem, the real concern is with the program exclusive of its library files and other program points from CodeSurfer representations. The purpose here is to find out how efficient the Greedy algorithm is in its decomposition of the program into a set of slices.

In Table 6, the percentage indicates the ratio of the size of the optimal set of slices to the size of the whole set of slices. The execution time shows running time to decompose the whole program. For each program, each function is decomposed into a set of slices using the Greedy algorithm. Figure 19 shows the frequency distributions of the percentage of the functions decomposed in each program.

As this figure shows, the majority of decomposed functions falls into the range where the percentage is lower than 25%. Many of them lie between 6% and 20% and relatively few are higher than 50%. This suggests that less than one fifth of a program can usually be used to decompose the whole program or function. Moreover, in Table 6, the percentage for the whole program also suggests that fewer than 20% of the program points can be used to decompose


(a) termutils (b) acct

(c) space (d) oracolo2

(e) byacc (f) a2ps

Fig. 19. Frequency distributions of the percentage of decomposed functions. The X axis is the percentage of the ratio of the size of the optimal set of slices to the size of all slices in a function; the Y axis is the number of decomposed functions which lie within the corresponding percentage ranges.

Programs	termutils2.0	acct6.3	space	oracolo2	byacc1.9	a2ps4.1
Size (Loc)	6697	9536	9126	14326	6337	42600
Number of vertices	11037	21382	20556	20551	33022	43141
Number of slices	2952	5305	9887	8776	8046	17226
Number of functions	56	88	137	135	178	248

Table 5
Program descriptions.

Programs	termutils2.0	acct6.3	space	oracolo2	byacc1.9	a2ps4.1
Percentage	14	20	18	13	19	20
Execution time of the program	6.7	44.0	126.4	105.0	140.4	1399.0
Execution time of the functions	1.4	3.3	8.3	8.1	7.5	208.4

Table 6
Percentage and execution time. Percentage is the ratio of the optimal set of slices to the set of all the slices in the whole program; execution time measured in seconds was obtained from execution on a Pentium4 3.2GHz with 512Mb RAM.

the whole program, capturing the programs semantics in the transitive closure of the dependencies of only one fifth of its program points.

In this empirical study, the non-Greedy algorithms are not considered. This is because it was found earlier that the Greedy approach outperforms the others. On the other hand, there are scalability issues for the Genetic, HC and Random algorithms with large programs. For instance, our experiments implement the 4 algorithms with the program *space*, a popular source code that has 9,887 source code program points. Except for the Greedy Algorithm, which runs in 126.4 seconds, the running times of the others are roughly 1 minute for each individual fitness function evaluation. That is, for both 100 populations and 100 generations, running times are about a week. This is clearly impractical.

5.1 Redundancy

This section reports the results of an experiment which aims to establish the extent to which redundancy in the slices affects the accuracy of decomposition. Redundancy here refers to instances when slices are the same as others or are completely included within others. Such slices are redundant because they will not contribute the fitness function value; they could always be subsumed by

Programs	sum	hello	informationflow	acct	newton	tss
Percentage	3.57	2.17	21.54	26.52	41.98	42.76
Programs	termutils-2.0	acct-6.3	space	oracolo2	byacc1.9	a2ps-4.1
Percentage	43.1	45.2	40.5	40.9	44.9	48.7

Table 7
Percentage of redundant slices. Slicing criteria are referred to as every program point in the programs.

an alternative slice.

Possible ways in which such redundancy would likely affect our search algorithms are as follows:

with the Greedy Algorithm: Firstly, where a slice is completely the same as another one or completely included by another one, the algorithm selects either one or the other (but not both). Secondly, even where slices are similar as opposed to exactly the same or included in others, the 'Greedy Strategy' enables the Greedy algorithm to take into consideration the extent of overlap (as well as of the coverage) when selecting - or not selecting - a given slice. Therefore, every slice chosen by the algorithm is one that will include the greatest fitness improvement. That is, the 'Greedy Strategy' will not select slices that are redundant.

with the GA and HC Algorithms: Although many identical or similar pairs of slices are likely to be selected as the initial generation, the fitness function values will gradually improve - through crossover and mutation (in the GA), or consideration of neighbours (in the HC). That is, redundancy will be filtered out as the algorithm progresses.

with the Random Algorithm: Redundancy could affect the algorithm's performance. However, the random algorithm is only considered as a baseline to check the performance of the other algorithms, rather than as a genuinely 'intelligent' search approach.

As the above analysis reveals, redundancy will not affect the value of fitness function with the Greedy, Genetic and HC algorithms, but it could affect overall execution times, especially when there are a lot of redundant slices in the program.

Table 7 lists the percentage of redundant slices of the 12 programs in our experiments. Redundancy is defined as the number of slices which are included in others as a percentage of the number of all the slices in the program. As shown in the figure, redundancy is prevalent in all 12 programs. It is suggested that execution time could be improved by reducing the redundancy in the source code. This is referred to as the further work.

6 Related Work

The work reported here is related to work in three areas that the paper draws together: set cover problems, decomposition slicing and Search Based Software Engineering (SBSE). This section briefly reviews related work on each of these three areas.

The set cover problem [31] is that of finding a minimally sized subset, σ of a set of sets, Σ, such that the distributed unions of σ and Σ are identical. The idea is that σ 'covers' the elements of Σ in the most economical manner. Though the set cover problem is NP-hard, it has been shown that greedy algorithms can provide good approximate answers [41]. This finding is also bourne out here for the application of set cover to slice decomposition.

The closest work to that reported in the present paper is that of Leon et al. [54], which considered minimum set cover in order to generate test cases by filtering techniques. The primary difference to the work reported here is that our set cover problem is also constrained by the need to reduce the overlap between the sets in the solution and, of course, the application area for the present paper is dependence analysis, not testing.

Gallagher introduced decomposition slicing, which captures all computation on a given variable [32]. Its purpose is to form a slice-based decomposition for programs through a group of algorithms. Decomposition slicing introduced a new software maintenance process model such that the need for regression testing can be eliminated. The idea is that changes to the decomposition slice that do not effect the compliment can be performed safely. In this way, the approach of limiting the side effects of software changes [33] has recently been extended by Tonella [73] to provide a mechanism for comprehension and impact analysis, using a formal concept lattice of slices.

The work reported here takes a different approach to decomposition. Rather than focusing on a particular variable, the approach seeks to find sets of criteria that partition the program. However, this was merely selected as one illustrative example of the general approach of search based slicing. One attractive aspect of this approach is the component-based nature of the search criteria; the fitness function is all that need be changed to search for a different set of slice properties.

The work reported here is an instance of Search Based Software Engineering (SBSE). Search based techniques have been applied to many software engineering activities from requirements engineering [5], project planning and cost estimation [1,3,4,20,27,48] through testing [7,8,15,18,19,37,39,55,59,74], to automated maintenance [16,30,38,61,62,64,71,72], service-oriented software engineering [22], compiler optimization [24,25] and quality assessment [17,47].

Although there has been much previous work on SBSE [29,39,57,62,74]. However, to the authors' knowledge, this is the first application of SBSE to dependence analysis.

In the last decade, several clone detection techniques have been investigated to detect duplicated code in programs exceeding hundreds of thousands lines of code: string-based [28,45], which is best suited for a first crude overview of the duplicated code; token-based [46,6], which works best in combination with a refactoring tool that is able to remove duplicated subroutines; parse-tree based [50,58,9], works best in combination with more fine-grained refactoring tools that work the statement level.

Moreover, Krinke [51] and Komondoor and Horwitz [49] introduce dependence-based approach to the identification of clone code by looking for similar isomorphic subgraphs of Procedure Dependence Graphs(PDG). The former constructs isomorphic subgraphs by partitioning the edges and vertices into the equivalence classes in terms of their PDG [44]. The latter constructs the similar subgraphs with backward and forward program slicing based on the PDG which also is partitioned into equivalence classes according to the syntactic structure of the statements and predicates of the program.

The observation of pictorial similarity in visualization of the optimal set of slices in Section 4.3 indicates that search based techniques for finding dependence patterns may also be useful in clone detection. More work is required to evaluate this possibility.

7 Conclusions

This paper has introduced a general framework for search based slicing, in which the principles of Search Based Software Engineering are used to formulate a problem of locating dependence structures as a search problem.

The paper presented results from an instantiation of this general framework of search based slicing, for the problem of program decomposition, presenting the results of a case study that evaluated the application of Greedy, Hill Climbing and Genetic Algorithms for both performance and similarity of results. Based upon the greedy algorithm, the best of 4 algorithms, further empirical study is formed to explore how efficiently large programs and single function can be decomposed.

The results indicate that the algorithms produce relatively consistent results and that the greedy algorithm outperforms its rivals. The results also provide the evidence that the landscape for the problem is either of low modality or is multi-modal, but with similarly valued peaks. The results are encouraging, because they suggest that it is possible to formulate dependence analysis problems as search problems and to find good solutions in reasonable time using this approach.

References

[1] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, Dec. 2001.

[2] B. W. B. andJ. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, , and M. J. Merritt. Characteristics of software quality. In *TRW and North-Holland Publishing Co.*, 1978.

[3] G. Antoniol, M. Di Penta, and M. Harman. A robust search–based approach to project management in the presence of abandonment, rework, error and uncertainty. In 10^{th} *International Software Metrics Symposium (METRICS 2004)*, pages 172–183, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[4] G. Antoniol, M. D. Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 240–249, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[5] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, Dec. 2001.

[6] B. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering 1995*, 1995.

[7] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[8] A. Baresel, H. Sthamer, and M. 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.

[9] I. Baxter, A. Yahin, L. Moura, and M. S. Anna. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, 1998.

[10] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[11] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21st *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[12] D. W. Binkley. The application of program slicing to regression testing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):583–594, 1998.

[13] D. W. Binkley and M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering*, 30(11):715–735, 2004.

[14] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[15] L. 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.

[16] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[17] S. Bouktif, H. Sahraoui, and G. Antoniol. Simulated annealing for improving software quality prediction. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1893–1900, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[18] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *SEKE*, pages 43–50, 2002.

[19] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028. ACM, 2005.

[20] C. J. Burgess and M. Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology*, 43(14):863–873, Dec. 2001.

[21] G. Canfora, A. Cimitile, A. De Lucia, and G. A. D. Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance*, pages 424–433, Los Alamitos, California, USA, Sept. 1994. IEEE Computer Society Press.

[22] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An approach for qoS-aware service composition based on genetic algorithms. In H.-G. Beyer and

U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1069–1075. ACM, 2005.

[23] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003.

[24] M. Cohen, S. B. Kooi, and W. Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1901–1908, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[25] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, volume 34.7 of *ACM Sigplan Notices*, pages 1–9, NY, May 5 1999. ACM Press.

[26] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Los Alamitos, California, USA, Mar. 1996. IEEE Computer Society Press.

[27] J. J. Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, 2000.

[28] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, 1999.

[29] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 65–74, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[30] D. Fatiregun, M. Harman, and R. Hierons. Search-based amorphous slicing. In 12^{th} *International Working Conference on Reverse Engineering (WCRE 05)*, pages 3–12, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Nov. 2005.

[31] U. Feige and P. Tetali. Approximating min sum set cover. In *Algorithmica*, volume 40, pages 219–234. Springer New York, Sept. 2004.

[32] K. B. Gallagher. *Using program slicing in software maintenance.* PhD thesis, University of Maryland, College Park, Maryland, Jan. 1990.

[33] K. B. Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*, pages 236–244, Los Alamitos, California, USA, Nov. 1992. IEEE Computer Society Press.

[34] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[35] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[36] Grammatech Inc. The codesurfer slicing system, 2002.

[37] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Constructing multiple unique input/output sequences using evolutionary optimisation techniques. *IEE Proceedings — Software*, 152(3):127–140, 2005.

[38] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[39] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[40] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[41] D. S. Hochbaum. Approximation algorithms for NP-hard problems. *PWS Publishing*, 1997.

[42] S. Horwitz, J. Prins, and T. Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[43] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[44] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[45] J. Johnson. Identifying redundancy in source code using fingerprints. In *Cascon*, 1993.

[46] T. Kamiya, S. Kusumoto, and K. I. Ccfinder. A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Engineering*, 28(7):654–670, 2002.

[47] T. M. Khoshgoftaar, L. Yi, and N. Seliya. A multiobjective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation*, 8(6):593– 608, December 2004.

[48] C. Kirsopp, M. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[49] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–??, 2001.

[50] K. Kontogiannis, R. Demori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Automated Software Engineering,*, 3(1), 1996.

[51] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[52] J. Krinke. Evaluating context-sensitive slicing and chopping. In *IEEE International Conference on Software Maintenance*, pages 22–31, Los Alamitos, California, USA, Oct. 2002. IEEE Computer Society Press.

[53] A. Lakhotia and J.-C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):677–689, 1998.

[54] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. pages 412–421, New York, NY, USA, 2005. ACM Press.

[55] Z. Li, M. Harman, and R. Hierons. Meta-heuristic search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering.* To appear.

[56] H. D. Longworth, L. M. Ott, and M. R. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)*, pages 383–389, 1986.

[57] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance*, pages 315–324, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[58] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, 1996.

[59] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24, Portland, Maine, USA., 2006.

[60] T. Meyers and D. W. Binkley. Slice-based cohesion metrics and software intervention. In 11^{th} *IEEE Working Conference on Reverse Engineering*, pages

256–266, Los Alamitos, California, USA, Nov. 2004. IEEE Computer Society Press.

[61] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1375–1382, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[62] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[63] H. Naeimi and A. DeHon. A greedy algorithm for tolerating defective crosspoints in NanoPLA design. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT2004)*, pages 49–56, Dec. 2004.

[64] M. O'Keeffe and M. OCinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, Mar. 2006.

[65] L. M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium*, pages 16–23, 1992.

[66] L. M. Ott and J. M. Bieman. Effects of software changes on module cohesion. In *IEEE Conference on Software Maintenance*, pages 345–353, Nov. 1992.

[67] L. M. Ott and J. M. Bieman. Program slices as an abstraction for cohesion measurement. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):681–699, 1998.

[68] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM Conference on Software Engineering*, pages 198–204, May 1989.

[69] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Los Alamitos, California, USA, May 1993. IEEE Computer Society Press.

[70] F. V. Rysselbergh and S. Demeye. Evaluating clone detection techniques. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Applications*, 2003.

[71] O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based improvement of subsystem decompositions. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1045–1051. ACM, 2005.

[72] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[73] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, 2003.

[74] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[75] M. Weiser. Program slicing. In 5^{th} *International Conference on Software Engineering*, pages 439–449, San Diego, CA, Mar. 1981.

[76] M. Weiser. Reconstructing sequential behaviour from parallel behaviour projections. *Information Processing Letters*, 17(10):129–135, 1983.

[77] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[78] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[79] D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.