Empirical Study of Fitness Functions for Search Based Slicing

MSc Advanced Software Engineering 2006
School of Physical Science & Engineering
King’s College London

Supervised by Professor Mark Harman
Empirical study of fitness functions for Search Based Slicing

Nan Wu

September 15, 2006
Abstract

This project focuses on the issues of researching the different kinds of fitness function for search-based program slicing using search algorithms. Because there are hundreds of and thousands of program lines, which always take too much time to read and understand them, the technique which is called program slicing, has to be used to divide large program into different parts based on the different program points. Therefore we can easy to understand them and reduce time and cost.

This project report attempts to analyse the rationale of the search algorithms, and then use them to implement the fitness function, such as fitness coverage and fitness overlap. Furthermore, comparing the value of different fitness function and performance of different search algorithms in different program size between

In order to achieve the goal, firstly, search algorithm should be understood very well. Secondly, try to find fitness functions in the different program size. Finally, use search algorithms to implement each fitness function in order to get the best fitness value.
Acknowledgements

There are many people that I would like to than in regards to their support and simply believing in me throughout my one year of studying.

Firstly, I would like to thank Professor Mark Harman for being a superb supervisor over the course of the project with guidance and assistance even extension time.

Secondly, Jiang Tao has provided me with valuable advice and technical support.

Furthermore, Shi Yoo and Lokuge Nadisha de Silva also gave some suggestion on the problem of report.

Finally, I would like to express thanks and love to my mother. When I have been despondent and sad because of the death of my father during writing this report, my mother controlled her sadness and had always given the support and confidence to me. I would not be where I am today if it were not for her.
Contents

1. Introduction ................................................. 8
   1.1 Aims and Objectives........................................... 8
   1.2 Brief Introduction of Main Related Techniques............. 8
   1.3 Structure of This Report....................................... 9

2. Literature Survey ........................................... 11
   2.1 Program Slicing............................................. 11
   2.2 Search Space................................................ 14
   2.3 Search-based Software Engineering........................... 15
   2.4 Fitness Function.............................................. 16
   2.5 Search-based Algorithms..................................... 17
       2.5.1 Genetic Algorithm....................................... 17
       2.5.2 Hill Climbing........................................... 22
       2.5.3 Randomize Algorithm.................................. 23

3. Design & Implementation Algorithms .......................... 24
   3.1 Fitness Function.............................................. 24
   3.2 Genetic Algorithms.......................................... 26
   3.3 Hill Climbing................................................ 27
   3.4 Randomize Algorithm......................................... 28
4. Evaluation

4.1 Results and Analysis...............................................................29

4.1.1 Search Algorithms in the Small-Sized Program..................30

4.1.2 Search Algorithms in the Medium-Sized Program.............33

4.1.3 Search Algorithms in the Large-Sized Program..................36

4.1.4 Search Algorithms in the Different Number of Generation.....37

5. Conclusion and Future Work..................................................42

5.1 Conclusion.................................................................42

6.2 Future Work...............................................................42

Bibliography.................................................................43

Appendices

Appendix 1    Source Code for Genetic Algorithms & Randomize Implementation

Appendix 2    Source Code for Hill Climbing Implementation
List of Figures

2.1 Example of a search space ........................................... 14
2.2 Process of GA. ..................................................... 18
2.3 Roulette Wheel. .................................................... 19
2.4 Graph of fitnesses before ranking ........................................ 19
2.5 Graph of fitnesses after ranking. .................................. 20
3.1 Fitness Coverage....................................................... 24
3.2 Fitness Overlap....................................................... 24
3.3 Pseudo-code for the Genetic Algorithms................................ 26
3.4 Pseudo-code for the Hill Climbing ................................... 28
3.5 Pseudo-code for the Randomized Algorithms....................... 28
4.1 WC................................................................. 30
4.2 Hello............................................................... 31
4.3 Computing.......................................................... 32
4.4 ACCT............................................................... 33
4.5 Fifth1.............................................................. 34
4.6 Fifth 2.............................................................. 35
4.7 Space............................................................... 36
4.8 Source file WC (the number of generation is 50) .................... 37
4.9 Source file WC (the number of generation is 100) .................. 37
4.10 Source file WC (the number of generation is 200) .................. 38
4.11 Source file Hello (the number of generation is 50) ................ 38
4.12 Source file Hello (the number of generation is 100) ................ 39
4.13 Source file Hello (the number of generation is 200) ............... 39
4.14 Source file ACCT (the number of generation is 50) ............... 40
4.15 Source file ACCT (the number of generation is 100) ............... 40
4.16 Source file ACCT (the number of generation is 200) ............... 41
Chapter 1

Introduction

This project aims to implement the different kinds of fitness function using search algorithms. This report formulates the problem as a search based software engineering problem.

1.1 Aims and Objectives

This project attempts to find different fitness function between the sets of program slices and then use search algorithms to analyze the results of each fitness function, arrive at optimal and near optimal solutions to constrained problems within large search spaces and finally compare what the different results and performance between each algorithms in small, medium and large program size. To evaluate the result, the report 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.

1.2 Brief Introduction of Main Related Techniques

The general explanation of related techniques will be introduced in this section.

Program slicing

Program slicing is a program analysis technique which can make the program clear, short and simple by focusing on selected program points. Program slicing can be divided into backward and forward slicing. A backward slice [1] contains the statements of the program which can have some effect on the slicing criterion, whereas a forward slice contains those statements of the program which are affected by the slicing criterion. In this project, backward slicing will be used.

Fitness function

In search-based software engineering, a fitness function is defined to capture a property of interest. The search process is implemented 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.
Genetic algorithms

Genetic algorithms are the part of evolutionary computing. GA is inspired by Darwin’s theory of evolution. Simply said, problems are solved by an evolutionary process resulting in a best solution. [2]

Hill climbing

In hill climbing, the search algorithms starts from a random starting point in the fitness landscape and tries to find one of its neighboring points which has a higher fitness value. If there is no neighboring point with higher fitness, the algorithm has reached the top of a hill.

Random

Normally, randomized algorithms seem to be the last methods to solve problems. If there are no feasible solution techniques, randomized algorithms are used in this case. Furthermore, they are always used in case the solution is so large that search is infeasible.

1.3 Structure of This Report

In this chapter, the aims and objectives of the project were presented. Moreover, the general introduction of main techniques was also explained. The remainder of this report is structured as follows.

Chapter 2 Literature Survey

This chapter provides the background and definitions of the related techniques, such as program slicing, search algorithms.

Chapter 3 Design & Implementation Algorithms

This chapter introduces the search algorithms used in the design of the system. The process designs of algorithms are mainly introduced through analyzing their functions. Also it focuses on introducing the search algorithms used in the design of the system. The key questions will be answered, such as “what does the code do in genetic algorithms/hill climbing/random?”

Chapter 4 Evaluation

This chapter provides detailed and graphical representations of the evaluation of the results obtained and a critical analysis about the evaluation made. Compare the fitness
function of different search algorithms from small to large program size.

Chapter 5 Conclusion and Future Work

The chapter summarizes the key achievements of the project. Furthermore, the uncompleted or unsatisfied results and performance are highlighted and discussed.
Chapter 2

Literature Survey

There are four main techniques which are closed related to this project. They are Program slicing, Search space, Search-based software engineering and Search-based algorithms. They will be explained and understood very well as follows.

2.1 Program Slicing

Program slice [3] consists of the parts of a program that affect the values computed at some point of interest, referred to as a slicing criterion. The task of computing program slices is called program slicing. Program slicing is an automated source code extraction technique that has been applied to several stages of the reverse engineering process, such as program comprehension[4, 5], and program integration[6]. Slicing can also be an effective way of understanding the dependence structure of a program and in measuring dependence-related attributes such as cohesion and coupling [7, 8]. Nowadays, there are some tools (commercial, scalable) for slicing such as Grammatech’s CodeSurfer [9] make it possible to construct many slices for a large program in reasonable time. It is now possible to slice a program at every possible program point, forming the set of slices that correspond to all possible slicing criteria. This allows slicing to be used to capture the dependence of every point in the program.

Harman and Hierons [10] have given us the definition of the program slicing. It is a technique for simplifying programs by focusing on selected aspects of semantics. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest. It is also a decomposition technique aimed at determining the subset of statements relevant to a computation of interest.

Furthermore, for example, if it consists of 10 program points in a program, there will be 10 slices, therefore, 1024 subsets of slices. Since the number of program points is always at least as large as the number of statements in the program, the power set of all possible slices will be extremely large. It is too large to enumerate for any realistically sized program. This observation motivates the search based approach introduced in the following section.

The program slicing provides the program statements which directly or indirectly contribute to the value of a given variable y at a given statement m. It can be expressed as $S(y; m)$. Harman said [11] A slice $S(y; m)$ of program P on variable y at
statement $m$ is a subprogram of $p$ computing the same value of $y$ at statement $m$ for every input on which $p$ terminates normally. In the following example, it is a source code which shows the difference between original source code and after slicing source code. The static slice is $S \, (\text{product, 13})$

**Original Source:**

```java
package examples;

public class MultiproceduralComputations{

    public void main(int n) {
        int i = 1;
        int sum = 0;
        int product = 1;
        while (i <= n) {
            sum = add(sum, i);
            product = multiply(product, i);
            i = add (i, 1);
        }
        System.out.println (sum);
        System.out.println (product);
    }

    int add (int a, int b) {
        return a+b;
    }

    int multiply (int c, int d) {
        int j = 1;
        int k = 0;
        while (j <= d) {
            k = add (k, c)
            j = add (j, 1);
        }
        return k;
    }
}
```

After slicing: $S$ (product, 13)

```java
package examples;

public class MultiproceduralComputations{

public void main(int n) {
    int i = 1;
    int product = 1;
    while (i <= n) {
        product = multiply(product, i);
        i = add(i, 1);
    }
    System.out.println(product);
}

int add (int a, int b) {
    return a+b;
}

int multiply (int c, int d) {
    int j = 1;
    int k = 0;
    while (j <= d) {
        k = add(k, c)
        j = add(j, 1);
    }
    return k;
}
```
2.2 Search Space

If we are solving a problem, we are usually looking for some solutions which will be the best among others. Marek obitko [2] have defined that the space of all feasible solutions is called search space. In Figure 1, it is an example of search space. Each point in the search space represents one possible solution. Each possible solution can be assessed by its value or fitness for solving the problem.

![Figure 2.1 Example of a search space](image)

The process of looking for a solution is the same as that of looking for some extreme value – minimum or maximum in the search space.

Also, search based approaches usually look for some solution that will be the best among others. The space of all feasible solutions consists of the set of solutions in which the optimum or near optimum solution resides is called search space. The purpose of all search algorithms is to locate the best solution among a number of possible solutions in the search space. In this project, the search space is the set of all the possible sets of slices based on the specific slicing statements. Each slice can get by extracting the program in terms of interest of the corresponding statement.

Suppose there is a program that has the $N$ number of statements, and $S_t$ represents a slice based on statement $t$ and $U$ represents the set of all the slices based on all the statements in the program, that is $U = \{S_1, S_2, S_3,...S_t,..., S_N\}$. The search space is the power set of $U$. 
2.3 Search-based Software Engineering

Before understanding search algorithms, the technique which is called search-based software engineering should be understood firstly.

“Search-based software engineering is a reformulation of software engineering as a search problem, in which the solution to a problem is found by sampling a large search space of possible solutions.” [12, 13]

For the past twenty years, engineers have started to know the search-based techniques, such as genetic algorithm, can be used in the fields of chemical, electrical, mechanical and so on. Because they can approach to optimal and near optimal results to constrained problems within large search spaces. Nowadays, search-based software engineering has experienced a rapid increase in activity and is a recent effort in the software engineering field which aims to apply the search techniques to the problems in software engineering. The search-based techniques now have different types of algorithms, which are from hill climbing to genetic algorithms.

The benefits of search-based techniques are:

1) It can deal with the problems which are too big to be handled by human hands.
2) The solutions which get from the search-based techniques are mostly based on numerical value.

Furthermore, Harman and Wegener [14] have observed the problems of software engineering as below:

- Competing constraints need to be balanced.
- We have to cope with inconsistency.
- There are many potential solutions.
- There is no perfect answer, but we still can recognize the good ones.

Moreover, Harman and Wegener also tell each participate who is with the ability to exploit search-based software engineering rationale and techniques four requirements as following:

- Genetic Algorithms (GA) and Hill Climbing (HC) are the two key search techniques, they should be understood.
Understanding the three major elements which can reformulate the existing problems of software engineering as search problems is needed: Representation, Fitness function and optimization techniques.

Understand how to determine whether the application of these techniques is effective.

2.4 Fitness Function

A fitness function [15] is a particular type in genetic algorithm of objective function that quantifies the optimality of a solution so that particular solution may be ranked against all other solution. It is the characterization of what is considered to be a good solution.

In search based software engineering, a fitness function is defined to capture a property of interest. In the case of search based slicing, it captures the properties of a dependence structure that makes it interesting to a particular analysis. The search process is implemented by an algorithm that uses the fitness function to guide a search that seeks to find optimal and near optimal solutions with respect to the fitness function. A good fitness function correlates closely with the algorithm’s goal.

There is generally no problem in determining the fitness function. In fact, most of the time, it is implicitly defined by the problem as we know what we want to optimize. But sometimes, particular attention should be taken due to the fact the selection is done according to the fitness of individuals. The fitness function should not only indicate how good solution is, but also it should correspond to how close the solution is to the optimal one.

The choice of a fitness function depends on 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.

There are two main fitness function which are closely related in this project, which are fitness coverage and fitness overlap. Fitness overlap has three features, which are average, maximum and tightness.

**Fitness coverage:** Calculate how much the program points in a slicing set can cover the program points of the whole program.

**Fitness overlap:** Calculate the number of program points of the intersection within a slicing set.
2.5 Search-based Algorithms

2.5.1 Genetic Algorithms

Genetic algorithms [16] are inspired by Darwin’s theory of evolution.

The process of evolution begins from the initial generation, with a set of solutions, which represented by chromosomes, called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected 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 its process of evolving candidates’ solutions towards the satisfied solution, genetic algorithm uses some operators, such as selection, encoding, crossover, mutation and replacement. GA also has some parameters, such as crossover probability, mutation probability and population size.

![Figure 2.2 Process of GA](image)

*Figure 2.2 Process of GA*

*Figure 2.2* shows the process and outline of basic GA. The details [marek obitko, Czech technical uni in Prague, 1998, introduction to genetic algorithms] of how does the GA works showed as follows.

1. [Start] Generate random population of n chromosomes
2. [Fitness] Evaluate the fitness of each chromosome in the population
3. [New population] Create a new population by repeating following steps until the new population is complete
   - [Selection] Select two parent chromosomes from a population according to their fitness
   - [Crossover] With a crossover probability, cross over the parents and form new offspring.
   - [Mutation] With a mutation probability mutate new offspring at each locus
   - [Accepting] Place new offspring in the new population
4. [Replace] Use new generated population for a further run of the algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in current population
6. [Loop] Go to step 2

The GA operators, such as selection, crossover and mutation have their own variations which are discussed below.

**Selection**

Determines which chromosomes are selected from the population to be parents for crossover based on their fitness. Also, it is the process of going from the current population to the next population constitutes one generation in the execution of a genetic algorithm.

Selection plays an important role in genetic algorithms because it is responsible for selecting the best individuals from the current population. If it does not take place successfully, genetic algorithms will not be able to obtain the optimal solution.

There are many methods in selecting the best chromosomes, such as roulette wheel, rank, and steady-state. They will be introduced below.

- **Roulette Wheel Selection**

  Roulette Wheel Selection [17] is the classic and more popular fitness-proportionate selection. It simply assigns to each solution a sector of a roulette wheel whose size is proportional to the appropriate fitness measurement, and then chooses a random position on the wheel.
In Figure 2.3, a marble is thrown in the roulette wheel and the chromosome where it stops is selected. The chromosomes with bigger fitness value will be selected more times. The percentages of an individual being accepted can be considered as

\[ p = \frac{\text{fitness score of the individual}}{\text{total fitness score of the population}} \times 100\% \]

For example, the probability of a in the population being accepted is

\[ a/(a+b+c+d+e+f+g) \times 100 = 1/(1+3+5+3+2+2+8) \times 100\% = 41.67\% \]

However, it will have problems when they are big difference between the fitness values. For example, if one chromosome fitness covers 90% of the sum of all fitnesses, then the other chromosomes will have few chances to be selected.

- Rank Selection

Rank Selection [marek obitko, Czech technical uni in Prague, 1998, introduction to genetic algorithms] ranks the population first and then every chromosome receives fitness value determined by this ranking.
Comparing *figure 2.4* and *figure 2.5*, all the chromosomes have a chance to be selected. But this method can not differ from the best chromosomes so much and others.

- **Steady-State Selection**

  The main idea of this type of selecting to the new population is that a big part of chromosomes can survive to next generation. The steady-state selection works in the following way. In every generation, a few good chromosomes are selected for creating new offspring. Then some bad chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation. This experiment uses the steady-state selection because it is better than other selection methods.

**Crossover**

Once the process of selection is completed, crossover takes place. Crossover could be considered as the process of creating the new population from the intermediate population by crossover parent individuals by applying one of the crossover methods. There are three types of crossover techniques used in this project, which are explained below. [2]

- **Single point crossover** – one crossover point is selected, binary string from the beginning of the chromosome to the crossover point is copied from the first parent, and the rest is copied from the other parent.
Two point crossover – two crossover points will be picked and the two strings will be split at these points.

Uniform crossover – bits are randomly copied from the first or from the second parent.
Mutation

After crossover has taken place, mutation operators are applied to the offspring. Each individual will be mutated in the new population when the probability of being mutated for that individual is less than a default probability. It can be described easily that selected bits are inverted.

\[ \begin{array}{c|c|c|c|c|c|c} & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \end{array} \]

After crossover

\[ \begin{array}{c|c|c|c|c|c|c} & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \hline & & & & & & \\ \end{array} \]

After mutation

11001001 \[\Rightarrow\] 10001001

*Figure 2.9 Mutation*

2.5.2 Hill Climbing

A hill climbing algorithm looks for the neighbour of current solutions and if the neighbour is better, this neighbour replaces the current solution. The operation will be repeated until no better neighbour can be found. In order to ensure that the hill climbing has the same cost of evaluating the fitness, the experiments uses the multi-point hill climbing to make sure each optimum solution with the hill climbing corresponding to one generation in the genetic algorithms.

Hill climbing [18], as the name suggests, begins its search from a random point in the search space and climbs towards a higher point in the space with an intention of climbing to the top of the hill. What happens in this algorithm is that it proceeds its search from a randomly chosen point and then compares itself with its neighbours, where it then makes a fitter neighbour the current state. This process continues until no fitter neighbour is found where it decides to terminate the process and regards the current state at the terminating instance as the maximum.

The hill climbing algorithm is always useful for the investigation of the fitness landscape.

Kirsopp [19] have applied the hill climbing algorithm to the optimization of a case-based reasoning system for the prediction of software project efforts, along with other algorithms. They found out that the fitness landscape contains a large scale structure with multiple points with different heights by comparing the results from the
hill climbing to those from the other search.

It can be described as follows:
- start with current state (initial state);
- until current state = goal state or there is no change in current state

2.5.3 Randomized Algorithm

A randomize algorithm [20] is an algorithm which employs a degree of randomness as part of its logic. In common practice, this means that the machine implementing the algorithm has access to a pseudo-random number generator. The algorithm typically uses the random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the “average case”. Formally, the algorithm’s performance will be a random variable determined by the random bits.

The purpose of using the randomize algorithm is to measure the performance of the other algorithms. This is because the randomize algorithm has an “intelligent”. Therefore, the performance of each other algorithm can be scaled by comparing it to the randomize algorithm. However, the same number of individuals, as that used by the genetic algorithms are needed in order to make sure the cost of evaluating the fitness in genetic algorithms, hill climbing and random is equal. Randomize search is the method to provide base-line data.
Chapter 3

Design & Implementation Algorithms

The project contains three search algorithms, which are Genetic Algorithms, Hill Climbing and Randomize Algorithm. The two fitness function which are called fitness coverage and fitness overlap will be focused. Also, these three algorithms will be compared and decided which search algorithms will optimize. Following sections discuss the design and implementation overview of each search algorithm implemented.

3.1 Fitness function

Fitness function is a particular type in genetic algorithm of objective function that quantifies the optimality of a solution so that particular solution may be ranked against all other solution.

In this project, fitness coverage and fitness overlap which includes FitnessOverlapAverage, FitnessOverlapMaximum and FitnessOverlapTightness. These fitness function is a parameter to the overall approach to search based slicing.

![Fitness Value = 7](image1)

*Figure 3.1 Fitness Coverage*

![Fitness Value = 2](image2)

*Figure 3.2 Fitness Overlap*
In order to explain what these fitness function stand for, the mathematical method has been chosen to illustrate them. For notational convenience, the following notation will be used.

(PHD student named Tao Jiang gave a lot of help and support in studying how to define the fitness function by mathematical methods).

\[ \cap (S_1, \ldots, S_i): \text{express the overlap of the } i \text{ number of slices;} \]
\[ \cup (S_1, \ldots, S_i): \text{express the union of the } i \text{ number of slices;} \]
\[ \text{Max}(S_1, \ldots, S_i): \text{express the largest slice in the } i \text{ number of slices that includes the most program points (} 1 < i < M). \]

FitnessCoverage - the percentages of the program points in a slicing cover the whole program points.

\[ \frac{\cup(S_1, \ldots, S_P)}{\cup(S_1, \ldots, S_M)} \times 100 \quad (1 < P < M) \]

FitnessOverlap - Evaluate the number of program points of the intersection in a slicing set.

Average: for each pair of slices, calculate the percentage of program points that are in both. Average value is evaluated based on all such pair.

\[ \frac{\cap(S_i, S_j)}{\text{Max}(S_i, S_j)} \times 100 \quad (1 < i < M) \]

Maximum: for each pair of slices, evaluate the percentage of program points that are in both. The maximum value is the largest value of the percentage of program points among all pair.

\[ \text{Max} \{ \frac{\cap(S_i, S_j)}{\text{Max}(S_i, S_j)} \} \times 100 \quad (0 < i \neq j < M) \]

Tightness: for all slices, find the program points which belong to all of the slices.

\[ \frac{\cap(S_1, \ldots, S_M)}{\cup(S_1, \ldots, S_M)} \times 100 \]
3.2 Genetic Algorithms

Genetic Algorithms are being designed and implemented using GA operators, which are called selection, encoding, crossover and mutation. Genetic algorithms are analysed by explaining the source code of GA. The pseudo-code for the GA shows as follows Figure 3.3.

//Parameters: Population (P): 30-50; Generation (G): 100;

1. begin
2. \( t = 0 \)
3. while\((t < G)\) do
4. \hspace{1cm} begin
5. \hspace{2cm} \( t = 0 \)
6. \hspace{2cm} initiate \( P(t) \)
7. \hspace{2cm} evaluate \( F(t) \)
8. \hspace{2cm} while \((t < P)\) do
9. \hspace{3cm} begin
10. \hspace{4cm} \( t = t+1 \)
11. \hspace{4cm} select \( P(t) \) from \( P(t-1) \)
12. \hspace{4cm} crossover \( P(t) \) according to crossover rate
13. \hspace{4cm} mutate \( P(t) \) according to mutate rate
14. \hspace{4cm} evaluate \( F(t) \)
15. \hspace{3cm} end
16. \hspace{2cm} \( t = t+1 \)
17. \hspace{1cm} end
18. end

Figure 3.3 Pseudo-code for the Genetic Algorithms

Initial \( P(t) \) on line 6 is the process of encoding the chromosomes in binary. For example, chromosome = \( \{1, 1, 0, \ldots 0, 0, 1, 1, 0\} \), each gene represents a slice, “1” stands for the chromosome includes the slicing, and “0” means the chromosome do not include the slicing.

Evaluate \( F(t) \) on line 7 is the method to evaluate \( F\) (Fitness Function) value for every chromosome in the population.

The selection methods are used to select a mating pool out of the initial population generated. The statement \( select P(t) from P(t-1) \) on line 11 implements the Elitism and Rank selection methods.

Crossover \( P(t) \), this is the multi-point crossover method on line 12, which are
represented by a string of binary, and there are multi-converted points, such as 0 to 1 or 1 to 0, which is called crossover points. The crossover methods exists a functionality which takes place once the offspring are produced. This will be to check if the generated offspring are within the specified budget. If the offspring do not happen to be within the budget the respective offspring will be replaced by its parents.

The method called \( \text{mutate } P(t) \) is to process the mutation operator on the generated offspring. For each offspring the method generates a random probability and compares it to the default probability. If the generated value is greater than the default, it will process the mutation on the particular offspring. Otherwise, it will move on to the next offspring.

3.3 Hill Climbing

The hill climbing algorithm starts with a random point in the search space. The algorithm compares the fitness of its current position with those of neighbouring points in the search space in iteration, using an objective function to evaluate the fitness of the points. If one of the neighbours has better fitness values, the algorithm will take this neighbour as its new position. It repeats its process until there is no neighbour with higher fitness value.

The pseudo-code for the hill climbing showed as below. Some important methods has been explained after the statement.

```
//Parameter: Generation (G): 100; Neighbour (N): the number of neighbours are considered as the same as population in GA.
1. begin
2.  \( t = 0 \)
3.  while (\( t < G \)) do
4.      begin
5.        initiate individual(t) // Binary Encoding
6.        while (\( t < \) the number of all the slices) do
7.          begin
8.            look for N(t) // Convert the gene value I sequence from the first one
9.              while(true) do
10.                 begin
11.                   if(F(I(t) < F(N(t))))
12.                     begin
13.                       the neighbour replace current individual
14.                         \( t = 0 \)
15.                        break
```
16. \text{end}
17. 
18. \text{else look for next neighbour of I(t)}
19. \text{end}
20. \text{end}
21. \text{t = t+1}
22. \text{end}
23. \text{end}

\text{Figure 3.4 Pseudo-code for the Hill Climbing}

\text{3.4 Randomize Algorithms}

The Randomize Algorithm is an algorithm that can make calls to a random number generator during the execution the algorithm. The form will be like
\text{x := Random (a, b), where a, b are integers, a<=b}

// Parameter: Generation (G): 100; individual (i): 30-50;

1. \text{begin}
2. \text{t = 0}
3. \text{while (t < G) do}
4. \text{begin}
5. \text{t = 0}
6. \text{while(t < I) do}
7. \text{begin}
8. \text{initiate I(t) // Randomly generate integer and implement the operation of module 2 to the integer}
9. \text{evaluate F(t)}
10. \text{t = t+1}
11. \text{end}
12. \text{t = t+1}
13. \text{end}
14. \text{end}

\text{Figure 3.5 Pseudo-code for the Randomized Algorithms}
Chapter 4

Evaluation

The evaluation of the search algorithms discussed in this project was carried out in two different ways which involved the analysis of result and performance of the algorithms. These two of measures are equally vital in the process of evaluating the algorithms implemented, as they examined different aspects of them. Each of the way of measures of analysis become quite important as they all contribute towards evaluating the algorithms implemented in different ways. It is difficult to say which algorithm is better than the other in evaluating. However, one could matter more than the other depending on the situation of the problem.

4.1 Results and Analysis

This chapter compares and illustrates the results and performance of three algorithms, such as Genetic Algorithms, Hill Climbing and Randomize Algorithm. These three search algorithms have all tried to get the best results and performance when they search the same fitness function (Fitness Coverage and Fitness Overlap).

There are two ways to compare the performance between these three search algorithms in the same search space and fitness function. One is that compare the three search algorithms in the different size of program codes from small to large source code. The other one is that compares the three search algorithms in the different number of generations from 30 to 100.

In fact, the process of comparing the three search algorithms is the similar as that of comparing the difference between Genetic Algorithms and the other two. Because the GA operators called crossover is the main difference between genetic algorithm and the other applicable algorithms and plays a key role in GA. Hill climbing algorithm often solve such problems in a smaller number of function evaluations than a genetic algorithm. Instead, identifying properties of spaces are easy for genetic algorithms and hard for hill climbing.

In the below section, the graphs are represented to illustrate the different fitness value and performance between these three algorithms in the same source code. The sizes of source codes are from small (around 50 program points) to large (more than 10,000 program points) and also the number of generation is increased in turn, so that the performance of each search algorithm can fully compared.
4.1.1 Search Algorithms in the Small-Sized Program

(For all graphs: x-axis is the number of generation; y-axis is the fitness value)

![Graph showing search algorithms performance](image)

**Figure 4.1 WC**

**Source file WC description:**

- The number of Program Lines: 37
- The number of Program Points: 57
- The number of Population: 30
- The number of Generation: 100
**Source file Hello description:**

- The number of Program Lines: 43
- The number of Program Points: 76
- The number of Population: 50
- The number of Generation: 100
**Source file Computing description:**

- The number of Program Lines: 148
- The number of Program Points: 326
- The number of Population: 100
- The number of Generation: 100
From the above graphs (Figure 4.1, Figure 4.2 and Figure 4.3) showed, the fitness value and performance are different between each search algorithm in the small program. There are some features of each algorithm through reading the graphs:

- The fitness values of Genetic Algorithm are always better or equal than the Hill Climbing and Random Algorithms.

- The fitness values of Random are better than the Hill Climbing at the most of the generation.

- The fewer number of program lines, more frequent the fitness value changed, otherwise not.

- As the increasing number of program lines, the fitness value of each algorithm decreased except the Worst-Based-on-GA, it increased.

### 4.1.2 Search Algorithms in the Medium-Sized Program

![Figure 4.4 ACCT](image)
Source file *ACCT* description:

- The number of Program Lines: 222
- The number of Program Points: 468
- The number of Population: 100
- The number of Generation: 100

![Chart](image)

*Figure 4.5 Fifth1*

Source file *Fifth1* description:

- The number of Program Lines: 362
- The number of Program Points: 998
- The Number of Population: 100
- The number of Generation: 100
Source file *Fifth2* description:

- The number of Program Lines: 521
- The number of Program Points: 1008
- The number of Population: 100
- The number of Generation: 100

From the above graphs (*Figure 4.4, Figure 4.5 and Figure 4.6*) showed, the features which were summarized on previous part (section 4.1.1) are the similar in the medium-size program.

- The fitness values of Genetic Algorithm are always better or equal than the Hill Climbing and Random Algorithms.
- The fitness values of Random are better than the Hill Climbing at the most of the generation.
- The more numbers of program lines, more infrequent the fitness value changed,
otherwise not. But for the Worst-Based-on-GA in the program fifth1, it has more continually changed in almost every generation.

4.1.3 Search Algorithms in the Large-Sized Program

![Figure 4.7 Space](image)

**Source file Space description:**

- The number of Program Lines: 9216
- The number of Program Points: 10003
- The number of Population: 100
- The number of Generation: 100

As the Figure 4.7 showed, when the number of program points achieved at a maximum value, the fitness values of all search algorithms keep the same as the initial value. However, due to the large program points, it will take so much time and cost to run them. Therefore, it may have some mistakes occurred during the previous running. That is reason why the graph looks like unbelievable or untrue. It should be corrected after several running times in the future.
4.1.4 Search Algorithms in the Different Number of Generation

In the different number of generation, the fitness value and performance of each search algorithms are different. The 50, 100, and 200 generation in the source files which are called WC, ACCT and Hello have been analyzed.

Source file WC

Figure 4.8 Source file WC (the number of generation is 50)

Figure 4.9 Source file WC (the number of generation is 100)
The three figures above showed the different number of generation for the WC source file. Obviously, the fitness values of GA in the different number of generation are better than the other search algorithms. In the first 50 generation, the fitness values of most of the search algorithms are gradually increased smoothly except Worse-Base-on-GA, which is fluctuant. In the next 150 generation, they do not change too much.

**Source file Hello**

*Figure 4.10 Source file WC (the number of generation is 200)*

*Figure 4.11 Source file Hello (the number of generation is 50)*
As the above three figure showed, the fitness value of GA are better than the other search algorithms in the first 50 generation and the fitness value of Worse-Based-on-GA still goes up-and-down frequently. In the second 50 generation, the fitness value of GA is no longer the maximum value among others, the
randomized algorithm had also achieved the maximum fitness value and get the better performance. Furthermore, the fitness value of hill climbing has arrived at the highest value in the last 50 generation. The fitness value and performance of GA still keep best.

**Source file ACCT**

**Figure 4.14 Source file ACCT (the number of generation is 50)**

**Figure 4.15 Source file ACCT (the number of generation is 100)**
Figure 4.16 Source file ACCT (the number of generation is 200)

The graphs showed above are the similar performance and fitness value with previous two source file WC and Hello.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

This project has discussed about how the search-based software engineering works. It provides a brand new tool which can reformulate software engineering problems as search problems. It can be used in the fields of chemical, electrical, mechanical and so on. Nowadays, the fields of using search-based software engineering are rapidly increasing.

In this project, the two fitness function, which are called fitness Coverage and Fitness Overlap have been defined by mathematical methods. Furthermore, compared the different search based algorithms, such as Genetic Algorithms and Heuristic Search Algorithms by fitness value and performance.

The most important contribution to this project is that illustrate the difference between Genetic Algorithms, Hill Climbing and Randomized Algorithm by graphs in different program size, which is from small to large program points and different number of generation. Therefore, Genetic Algorithms have been regarded as that offering much potential and advantages over other algorithms.

5.2 Future Work

In chapter 4, when evaluated the large program, which is more than 10,000 program points, more times of running should be taken so that the unbelievable results can be confirmed whether it is reasonable or not. Even try to run the source code with many more program points.

Also, with the availability of more time, it would be had been possible to find many more fitness function.
Bibliography

[1] Mark Harman and Robert m. hierons, an overview of program slicing, http: www.brunel.ac.uk/-csstmmh2


[10] An overview of program slicing, mark Harman and Robert m. hierons, Brunel University


[14] Mark Harman and Joachim Wegener, Getting Results from Search-Based Approaches to Software Engineering


[18] Mark Harman and Robert Hierons, Reformulating software engineering as a search problem


Appendices

Appendix 1  Source Code for GA & Randomize Implementation

Appendix 2  Source Code for Hill Climbing Implementation
Appendix 1  Source Code for GA & Randomize Implementation
package uk.ac.kcl.dcs.nanwu;

import java.util.Random;
import java.util.StringTokenizer;
import java.io.*;
import java.math.*;

public class gaslicing {
    private int Population;
    private int SourceFileLine;
    private int VertexSet = 177;
    private float MutationProbability, CrossOverProbability;
    private int SlicingMatrix[][] = new int[1000][1000];
    private int SolutionMatrix[][] = new int[1000][1000];
    private int FitnessIntersection[] = new int[1000];
    private int FitnessOverlapLarge[] = new int[1000];
    private int FitnessOverlapAverage[] = new int[1000];
    private int FitnessCoefficient[] = new int[1000];
    private int FitnessCover[] = new int[1000];
    private int CrossOverPoint[][] = new int[1000][1000];
    private int value[] = new int[1000];
    private int Slicing[][] = new int[1000][1000];
    int count = 0;
    static int test;
    int test22;

    // tokenizer
    public static void main(String[] args) throws IOException {


gaslicing test = new gaslicing(1000, (float) 0.8, (float) 0.5, 66, 177);
test.testToken();
test.loadFile();
test.value();
test.nonempty();
test.InitialRandomPopulation();

int n = 0;
boolean k = true;
while (k) {
    for (int i = 0; i < 3; i++)
        System.out.println("*******************************************************************
***************************
*******************************************************************
*****************
* * * ");
}

System.out.println("*******************************************************************
***************************
*******************************************************************
********************
Population("
    + n
    + "")
    + "***");
for (int i = 0; i < 3; i++)
    System.out.println("*******************************************************************
***************************
*******************************************************************
********************");

for (int i = 0; i < test.SourceFileLine; i++) {
    // test.display();
    // test.InitialRandomPopulation();
    test.GenerateCrossOverPoint();
test.Crossover();
test.Mutation();
for (int indx = 0; indx < test.SouceFileLine; indx++) {
    test.FitnessCover[indx] = test.FitnessCover(indx);
    test.FitnessIntersection[indx] = test
        .FitnessIntersection(indx);
    test.FitnessOverlapAverage[indx] = test
        .FitnessOverlapAverage(indx);
    test.FitnessOverlapLarge[indx] = test
        .FitnessOverlapLarge(indx);
    test.FitnessCoefficient[indx] = test
        .DefineFitnessCoefficient((float) 0.5, (float) 0.5,
        indx);
    System.out.print("Solution" + indx + "------FitnessCover:
    + test.FitnessCover[indx] + "/176 ");
    System.out.print("------FitnessIntersection:
    + test.FitnessIntersection[indx] + ");
    System.out.print("------FitnessOverlapAverage:
    + test.FitnessOverlapAverage[indx] + ");
    System.out.print("------FitnessOverlapLarge:
    + test.FitnessOverlapLarge[indx] + ");
    System.out.println("------FitnessCoefficient:
    + test.FitnessCoefficient[indx] + ");
    System.out.println("-----------------------------------------------------------------------
    + " + "---------------------------------------------------------------");
}
// for(int i=0;i<test.SouceFileLine;i++){
// System.out.print("Solution"+i+"--FitnessCover:"+test.FitnessCover[i]+"
// ");
// System.out.print("Solution"+i+"--FitnessIntersection:"+test.FitnessIntersection[i]+"
// ");
// System.out.print("Solution"+i+"--FitnessOverlapAverage:"+test.FitnessOverlapAverage[i]+"
// ");
// System.out.print("Solution"+i+"--FitnessOverlapLarge:"+test.FitnessOverlapLarge[i]+"
// ");
// System.out.println("Solution"+i+"--FitnessCoefficient:"+test.FitnessCoefficient[i]+"
// ");
//}
System.out.println("Solution"+i+"--FitnessCoefficient:"+test.FitnessCoefficient[i]+" // ");

public void loadFile() {

    // 1. Reading input by lines:
    try {
        BufferedReader in = new BufferedReader(new FileReader("output.txt"));
        String s, s1;
        StringTokenizer token;
        int index = 0;
        while ((s = in.readLine()) != null) {
            token = new StringTokenizer(s);
            int i = 0;
            while (token.hasMoreTokens()) {
                s1 = token.nextToken();
                // System.out.println("s1="+s1);
                // s1 = "test";
                // SlicingMatrix[][]indx
                // System.out.println("Matrix["+i+"]["+index+"]=""+s1);
                SlicingMatrix[i][index] = Integer.parseInt(s1);

                // test = Integer.parseInt(s1);

            }
            try {
                BufferedReader in4 =
                        new BufferedReader(new StringReader("outfile.txt"));
                PrintWriter out1 =
                        new PrintWriter(new BufferedWriter(new FileWriter("analysis.txt")));
                int lineCount = 1;
                while ((s = in4.readLine()) != null)

            }
    }

}
i++;
// System.out.println("SlicingMatrix=" +
// SlicingMatrix[i][index]);
}
index++;
// System.out.println("SlicingMatrix=" +
// SlicingMatrix[i][index]);
}

// s2 += s + ",n";
// System.out.println(s + ",n");
in.close();

} catch (IOException ex) {

}

gaslicing(int nPopulation, float nCrossOverProbability,
float nMutationProbability, int nSourceFileLine, int nVertexSet) {

    Population = nPopulation;
    CrossOverProbability = nCrossOverProbability;
    MutationProbability = nMutationProbability;
    SourceFileLine = nSourceFileLine;
    VertexSet = 177;
}

public void ReadFromOutput() {

}
public void InitialRandomPopulation() {
    int random;
    for (int index = 0; index <= Population - 1; index++)
        for (int a = 0; a < SourceFileLine; a++) {
            random = Random();

            SolutionMatrix[a][index] = random % 2;
            // System.out.println(SolutionMatrix[a][index]);
        }
}

global int FitnessCover(int indx) {
    // int temp = 0
    int count1 = 0;
    int FitnessValue1 = 0;
    int tempMatrix[] = new int[1000];
    for (int i = 0; i < SourceFileLine; i++) {
        // int j = 0;
        if (SolutionMatrix[i][indx] == 1) {
            tempMatrix[count1] = i;
            // j++;
            count1++;
        }
    }

    for (int i = 1; i < VertexSet; i++) {
        int temp1 = 0;
        for (int j = 0; j < count1; j++) {
            // System.out.println(Slicing[i][tempMatrix[j]]);
            if (Slicing[i][tempMatrix[j]] == 1)
                temp1++;
        }
        if (temp1 != 0)
            FitnessValue1++;
    }
    return FitnessValue1;
}

global int FitnessIntersection(int indx) {
    int count2 = 0;
    int FitnessValue2 = 176;
    int tempMatrix[] = new int[1000];

for (int i = 0; i < SourceFileLine; i++) {
    if (SolutionMatrix[i][indx] == 1) {
        tempMatrix[count2] = i;
        count2++;
    }
}

for (int i = 1; i < VertexSet; i++) {
    int temp1 = 0;
    int zero = -1;
    for (int j = 0; j < count2; j++) {
        if (Slicing[i][tempMatrix[j]] == 0)
            temp1++;
    }
    // System.out.println("i\n"+i+" "+temp1);
    if (temp1 != 0)
        FitnessValue2--;
    else
        zero = i;
    if (zero != -1)
        System.out.println("Location of Intersection point: "+ zero);
}

return FitnessValue2;
}

public int FitnessOverlapAverage(int indx) {
    // 1. Overlap 1: Average
    // For each pair of slices: measure the percentage of nodes that are in
    // both. Take the average of all such pairwise comparisons.
    int SlicingSize[] = new int[1000];
    int Average[][] = new int[1000][1000];
    // int OverlapAverage[] = new int[1000];
    int AverageValue;
    // int temp = 0;
    int sum = 0;
    int count3 = 0;
    // int FitnessValue = 0;

    int tempMatrix[] = new int[1000];
    for (int i = 0; i < SourceFileLine; i++) {
        for (int j = 0; j < SourceFileLine; j++) {
            Average[i][j] = 0;
        }
    }
for (int i = 0; i < SourceFileLine; i++) {
    SlicingSize[i] = 0;
}

for (int i = 0; i < SourceFileLine; i++) {
    if (SolutionMatrix[i][indx] == 1) {
        tempMatrix[count3] = i;
        count3++;
    }
}

for (int i = 0; i < count3; i++) {
    // SlicingSize[tempMatrix[i]] = 0;
    for (int j = 1; j < VertexSet; j++) {
        if (Slicing[j][tempMatrix[i]] == 1) {
            SlicingSize[tempMatrix[i]]++;
        }
    }
}

for (int i = 0; i < count3 - 1; i++) {
    for (int a = i + 1; a < count3; a++) {
        for (int j = 1; j < VertexSet; j++)
            if ((Slicing[j][tempMatrix[i]] == Slicing[j][tempMatrix[a]])
                && (Slicing[j][tempMatrix[i]] == 1)
                && (tempMatrix[i] != tempMatrix[a]))
                Average[tempMatrix[i]][tempMatrix[a]]++;

        if (Math.min(SlicingSize[tempMatrix[i]],
                     SlicingSize[tempMatrix[a]]) != 0)
            Average[tempMatrix[i]][tempMatrix[a]] =
            100 * (Average[tempMatrix[i]][tempMatrix[a]] / Math
                     .min(SlicingSize[tempMatrix[i]],
                          SlicingSize[tempMatrix[a]])); // Evaluate

        // Average
        // Average[tempMatrix[i]][tempMatrix[a]] =
        // 100*Average[tempMatrix[i]][tempMatrix[a]]/(SlicingSize[tempMatrix[i]]+SlicingSize[tempMatrix[a]]);

        // System.out.println(11%10);
        sum = (Average[tempMatrix[i]][tempMatrix[a]] + sum);
        // System.out.println(SlicingSize[tempMatrix[i]]);
        // System.out.println(SlicingSize[tempMatrix[a]]);
System.out.println(Math.min(SlicingSize[tempMatrix[i]], SlicingSize[tempMatrix[a]]));
System.out.println("Average["+tempMatrix[i]+"]" +tempMatrix[a]+"]=" +Average[tempMatrix[i]] [tempMatrix[a]]);
System.out.println("n");
}
}

for (int i = 0; i < count3; i++) {
for (int j = 0; j < count3; j++) {
sum = Average[tempMatrix[i]][tempMatrix[i+j]]+sum;
}
}

AverageValue = sum * 2 / (count3 * (count3 - 1));
System.out.println(AverageValue);
return AverageValue;

public int FitnessOverlapLarge(int indx) {

// For each pair of slices: measure the percentage of nodes that are in
// both. Take the largest value as the final result for overlap.
int SlicingSize[] = new int[1000];
int Average1[][] = new int[1000][1000];
int OverlapLarge[] = new int[1000];
int LargeOverlap[][] = new int[1000][1000];
int value = 0;

// int temp = 0;
int count4 = 0;
// int FitnessValue = 0;
int tempMatrix[] = new int[1000];
for (int i = 0; i < SouceFileLine; i++) {
SlicingSize[i] = 0;
}
for (int i = 0; i < SouceFileLine; i++) {
for (int j = 0; j < SouceFileLine; j++) {
Average1[i][j] = 0;
}
}
for (int i = 0; i < SouceFileLine; i++) {
for (int j = 0; j < SourceFileLine; j++) {
    LargeOverlap[i][j] = 0;
}
}
for (int i = 0; i < SourceFileLine; i++) {
    if (SolutionMatrix[i][indx] == 1) {
        tempMatrix[count4] = i;
        count4++;
    }
}
}
for (int i = 0; i < count4; i++) {
    // SlicingSize[i] = 0;
    for (int j = 1; j < VertexSet; j++) {
        if (Slicing[j][tempMatrix[i]] == 1) {
            SlicingSize[tempMatrix[i]]++;
        }
    }
}
for (int i = 0; i < count4 - 1; i++) {
    for (int a = i + 1; a < count4; a++) {
        for (int j = 1; j < VertexSet; j++) {
            if ((Slicing[j][tempMatrix[i]] == Slicing[j][tempMatrix[a]])
                && (Slicing[j][tempMatrix[i]] == 1)
                && ((tempMatrix[i] != tempMatrix[a]))) {
                Average1[tempMatrix[i]][tempMatrix[a]]++;
                // System.out.println(Average1[tempMatrix[i]][tempMatrix[a]]);
            }
        }
        if (Math.min(SlicingSize[tempMatrix[i]],
                     SlicingSize[tempMatrix[a]]) != 0)
            Average1[tempMatrix[i]][tempMatrix[a]] = 100
                * Average1[tempMatrix[i]][tempMatrix[a]]
                / Math.min(SlicingSize[tempMatrix[i]],
                            SlicingSize[tempMatrix[a]]); // evaluate
            // LargeOverlap[tempMatrix[i]][tempMatrix[a]] =
            // Average1[tempMatrix[i]][tempMatrix[a]]; // evaluate Large
            // Average1[tempMatrix[i]][tempMatrix[a]] =
            // 100*Average1[tempMatrix[i]][tempMatrix[a]]/(SlicingSize[tempMatrix[i]]+SlicingSize[tempMatrix[a]]); //
evaluate
LargeOverlap[tempMatrix[i]][tempMatrix[a]] =
Average1[tempMatrix[i]][tempMatrix[a]]; // evaluate
// Large
System.out.println("Average1[" + tempMatrix[i] + "]="
+ tempMatrix[a] + "]="
+ Average1[tempMatrix[i]][tempMatrix[a]]);
System.out.println(LargeOverlap[tempMatrix[i]][tempMatrix[a]]);
if (LargeOverlap[tempMatrix[i]][tempMatrix[a]] > value
    && LargeOverlap[tempMatrix[i]][tempMatrix[a]] != 100)
    value = LargeOverlap[tempMatrix[i]][tempMatrix[a]];
}

// int Temp;
//
// for (int i = 0; i <= SourceFileLine - 1; i++) {
//     for (int j = 0; j < SourceFileLine - 2; j++)
//         if (LargeOverlap[i][j] < LargeOverlap[i][j + 1])
//             Temp = LargeOverlap[i][j];
//         LargeOverlap[i][j] = LargeOverlap[i][j + 1];
//     LargeOverlap[i][j + 1] = Temp;
//         
// }
// }

// for (int i = 0; i <= SourceFileLine - 2; i++) {
//     for (int j = 0; j < SourceFileLine - 1; j++)
//         if (LargeOverlap[j][i] < LargeOverlap[j][i + 1])
//             Temp = LargeOverlap[j][i];
//         LargeOverlap[j][i] = LargeOverlap[j][i + 1];
//     LargeOverlap[j][i + 1] = Temp;
//     
// }
// }

// return value;

public int DefineFitnessCoefficient(float FitnessOverlapAverageCoefficient,
float FitnessOverlapLargeCoefficient, int indx) {
    // combine these into a single fitness value by multiplying each by a
    // >coefficient. The coefficient says how much we "care" about each of
    // the
    // >two. To start with make this coefficient the same for both part of
    // the
    // >fitness calculation.

    FitnessCoefficient[indx] = (int) (FitnessOverlapAverageCoefficient
        * FitnessOverlapAverage[indx] + FitnessOverlapLargeCoefficient
        * FitnessOverlapLarge[indx]);

    return FitnessCoefficient[indx];
}

// public int FitnessOverlapLarge(int index){}
// public int FitnessOverlapAverage(int index){}
public void Crossover() {
    int TempMatrix[][] = new int[1000][2];
    int TempMatrix0[] = new int[1000];
    int TempMatrix1[] = new int[1000];
    int Temp, j, k;

    for (int index = 0; index <= (Population / 4) - 1; index++)
        for (int t = 0; t <= 1; t++) {

            for (int i = 0; i < SourceFileLine; i++)
                TempMatrix0[i] = SolutionMatrix[i][2 * index];
            TempMatrix1[i] = SolutionMatrix[i][2 * index + 1];

            for (int i = 0; i < SourceFileLine; i++)
                if (CrossOverPoint[i][2 * index + t] == 0) {
                    for (j = 0; j < SourceFileLine; j++)
                        if (TempMatrix0[j] != 100) {
                            TempMatrix[t][i] = TempMatrix0[j];
                            Temp = TempMatrix0[j];
                            TempMatrix0[j] = 100;
                            j = SourceFileLine - 1;

                            for (k = 0; k < SourceFileLine; k++)
                                if (TempMatrix1[k] == Temp) {
                                    TempMatrix1[k] = 100;
                                    k = SourceFileLine - 1;
                                }
                        }
                }
        }
}
else {
    for (j = 0; j < SourceFileLine; j++)
        if (TempMatrix1[j] != 100) {
            TempMatrix[i][t] = TempMatrix1[j];
            Temp = TempMatrix1[j];
            TempMatrix1[j] = 100;
            j = SourceFileLine - 1;

            for (k = 0; k < SourceFileLine; k++) {
                if (TempMatrix0[k] == Temp) {
                    TempMatrix0[k] = 100;
                    k = SourceFileLine - 1;
                }
            }
        }
}

for (int i = 0; i < SourceFileLine; i++)
    SolutionMatrix[i][2 * index + Population / 2 + t] = TempMatrix[i][t];

}

public void Mutation() {

    int randomChromosome;
    int randomGen0, randomGen1;
    int Temp;
    int NumberOfMutation = (int) (MutationProbability * (Population - 1) * SourceFileLine);

    for (int k = 0; k <= NumberOfMutation; k++) {
        randomChromosome = 0;
        while ((randomChromosome = Random() % Population) == 0)
            ;

        randomGen0 = Random() % SourceFileLine;
        while ((randomGen1 = Random() % SourceFileLine) == randomGen0)
            ;

        Temp = SolutionMatrix[randomGen0][randomChromosome];
```java
SolutionMatrix[randomGen0][randomChromosome] = SolutionMatrix[randomGen1][randomChromosome];
SolutionMatrix[randomGen0][randomChromosome] = Temp;
}
}

public void GenerateCrossOverPoint() {
    int randomCrossOver;
    for (int index = 0; index <= Population - 1; index++) {
        for (int a = 0; a < SourceFileLine; a++) {
            randomCrossOver = Random();
            // System.out.println(randomCrossOver);
            // System.out.println(randomCrossOver);
            CrossOverPoint[a][index] = randomCrossOver % 2;
            // System.out.print(CrossOverPoint[a][index]);
        }
    }
}

public void FitnessCoefficientOrder() {
    boolean k = true;
    int Temp;
    while (k) {
        k = false;
        for (int i = 0; i <= Population - 2; i++) {
            if (FitnessCoefficient[i] < FitnessCoefficient[i + 1]) {
                Temp = FitnessCoefficient[i];
                FitnessCoefficient[i] = FitnessCoefficient[i + 1];
                FitnessCoefficient[i + 1] = Temp;
                for (int j = 1; j < VertexSet; j++) {
                    Temp = Slicing[j][i];
                    Slicing[j][i] = Slicing[j][i + 1];
                    Slicing[j][i + 1] = Temp;
                }
                k = true;
            }
        }
    }
}
```
public int Random() {
    Random generator = new Random();
    int random;
    random = generator.nextInt(100);
    return random;
}

public int evaluate(int indx) {
    int value = 0;
    for (int i = 1; i < 177; i++)
        value = SlicingMatrix[i][indx] + value;
    return value;
}

public void nonempty() {
    for (int i = 0; i < 109; i++)
        if (value[i] != 0) {
            for (int j = 0; j < 177; j++)
                Slicing[j][count] = SlicingMatrix[j][i];
            count++;
        }
    // System.out.println(count);
}

public void value() {
    for (int i = 0; i < 109; i++)
        value[i] = evaluate(i);
}

public void display() {
    for (int i = 0; i < count; i++)
        int number = -1;
        for (int j = 0; j < 177; j++)
            System.out.print("Slicing[" + j + "][" + i + "] = " + Slicing[j][i] + ",");
        if (j == 23)
            number = Slicing[j][i];
}
public void testToken() {
    StringTokenizer token;
    String str = "1 2 3 4";
    token = new StringTokenizer(str);
    while (token.hasMoreTokens()) {
        str = token.nextToken();
        //System.out.println("s1=\""+str);
    }
}
}
Appendix 2 Source Code for Hill Climbing Implementation
package uk.ac.kcl.dcs.nanwu;

import java.util.*;
import java.util.StringTokenizer;
import java.io.*;

public class Gaslicing {

    private int SourceFileLine; // = 2000;
    private int VertexSet; // = 2000;
    private byte SlicingMatrix[][] = new byte[500][2000];
    private int SolutionMatrix[] = new int[1000];
    // private byte Neighbor1[] = new byte[1000];
    // // private byte Neighbor2[] = new byte[1000];
    private int FitnessOverlapAverage;
    private int FitnessCover;
    private int fitnessFinal;
    private int value[] = new int[2000];
    private byte Slicing[][] = new byte[500][300];
    private int count = 0;
    // static int test;
    // // int test22;

    // tokenizer
    public static void main(String[] args) throws IOException {
        Gaslicing test = new Gaslicing(20, 34);
        int bestvalue[] = new int[2000];
        bestvalue[0] = 15;
        int n = 1;
    }
}
int best = 0;
// test.testToken();
test.loadFile();
test.value();
test.nonempty();
while (n < 100) {
    test.InitialRandomPopulation();

    // test.FitnessCover = test.FitnessCover(test.SolutionMatrix);
    // test.FitnessOverlapAverage = test
    // .FitnessOverlapAverage(test.SolutionMatrix);
    // test.fitnessFinal = test
    // .fitnessFinal(test.SolutionMatrix, 0.5, 0.5);

    System.out
        .println("----------------------------------------------------------------------
        + " + "-------------------------------------------------------------
        + " + "-----------------------------------------");

    best = test.hillclimbing(test.SolutionMatrix);
    bestvalue[n] = bestvalue[n - 1];

    if (best > bestvalue[n])
        bestvalue[n] = best;

    System.out.println("Solution" + ":[" + n + "]" + "FitnessFunction="
    + best);

    // }

    n++;
}

System.out.print("FitnessEvaluation ");
for (int i = 0; i < n; i++)
    System.out.print(i + " ");

System.out.println();
System.out.print("HillClimbing ");
// System.out.print("Fitness-Based-on-Random ");
for (int i = 0; i < n; i++)
    System.out.print(bestvalue[i] + " ");
System.out.println();
public int hillclimbing(int solution[]) {  
    int solution1[] = new int[1000];  
    int bestvalue0 = 0;  
    int bestvalue1 = 0;  
    bestvalue0 = fitnessFinal(solution, 0.5, 0.5);  
    for (int i = 0; i < count; i++) {  
        solution1 = solution;  
        if (solution1[i] == 1)  
            solution1[i] = 1;  
        else solution1[i] = 0;  
        bestvalue1 = fitnessFinal(solution1, 0.5, 0.5);  
        if (bestvalue1 > bestvalue0) {  
            solution = solution1;  
            bestvalue0 = bestvalue1;  
            i = 0;  
        }  
    }  
    return bestvalue0;  
}

public void loadFile() {  
    // String s2="1";  
    // System.out.println(s2);  
    // 1. Reading input by lines:  
    try {  
        BufferedReader in = new BufferedReader(new FileReader("gasum.txt"));  
        String s, s1;  
        // System.out.println(s1);  
        StringTokenizer token;  
        int index = 0;  
        while ((s = in.readLine()) != null) {  
            token = new StringTokenizer(s);  
            int i = 0;  
            while (token.hasMoreTokens()) {  
                s1 = token.nextToken();  
                // System.out.println("Matrix[\"+i\"]\[\"+index+\"] = \"+s1);  
            }  
        }  
    } catch (Exception e) {  
        // Error while reading file.  
    }  
}
SlicingMatrix[i][index] = (byte) Integer.parseInt(s1);
// SlicingMatrix[i][index] = (byte).s1;

// test = Integer.parseInt(s1);

// try {
// BufferedReader in4 =
// new BufferedReader(
// new StringReader("outfile.txt"));
// PrintWriter out1 =
// new PrintWriter(
// new BufferedWriter(
// new FileWriter("analysis.txt"));
// int lineCount = 1;
// while((s = in4.readLine()) != null )
// out1.println(lineCount++ + ": " + s);
// out1.close();
// } catch(EOFException e) {
// System.err.println("End of stream");
// }

i++;
// System.out.println("SlicingMatrix= " +
// SlicingMatrix[i][index]);

} index++;
// System.out.println("SlicingMatrix= " +
// SlicingMatrix[i][index]);

// s2 += s + "\n";
// System.out.println(s + " ");
in.close();

} catch (IOException ex) {

}

}

Gaslicing(int nSourceFileLine, int nVertexSet) {

SourceFileLine = nSourceFileLine;
VertexSet = nVertexSet;

68
public void InitialRandomPopulation() {
    int random;
    for (int a = 0; a < count; a++) {
        random = Random();

        SolutionMatrix[a] = (random % 2);
        // System.out.println(SolutionMatrix[a][index]);
    }
}

public int FitnessCover(int matrix[]) {
    // int temp = 0
    int count1 = 0;
    int FitnessValue1 = 0;
    int tempMatrix[] = new int[2000];
    for (int i = 0; i < count; i++) {
        // int j = 0;
        if (matrix[i] == 1) {
            tempMatrix[count1] = i;
            // j++;
            count1++;
        }
    }

    for (int i = 0; i < VertexSet; i++) {
        int temp1 = 0;
        for (int j = 0; j < count1; j++) {
            // System.out.println(Slicing[i][tempMatrix[j]]);
            if (Slicing[i][tempMatrix[j]] == 1)
                temp1++;
        }
        if (temp1 != 0)
            FitnessValue1++;
    }
    FitnessValue1 = 100 * FitnessCover / VertexSet;
    return FitnessValue1;
}

public int FitnessOverlapAverage(int matrix[]) {
    // 1. Overlap 1: Average
// For each pair of slices: measure the percentage of nodes that are in
// both. Take the average of all such pairwise comparisons.
int SlicingSize[] = new int[2000];
byte Average[][] = new byte[1000][1000];
// int OverlapAverage[] = new int[10000];
int AverageValue = 100;
// int temp = 0;
int sum = 0;
int count3 = 0;
// int FitnessValue = 0;

int tempMatrix[] = new int[2000];
// for (int i = 0; i < count; i++) {
// for (int j = 0; j < count; j++) {
// Average[i][j] = 0;
// }
// }
// for (int i = 0; i < count; i++) {
// SlicingSize[i] = 0;
// }
for (int i = 0; i < count; i++) {
if (matrix[i] == 1) {
tempMatrix[count3] = i;
count3++;
}
}
for (int i = 0; i < count3; i++) {
// SlicingSize[tempMatrix[i]] = 0;
for (int j = 0; j < VertexSet; j++) {
if (Slicing[j][tempMatrix[i]] == 1) {
    SlicingSize[tempMatrix[i]]++;
}
}
}
for (int i = 0; i < count3 - 1; i++) {
for (int a = i + 1; a < count3; a++) {
    for (int j = 0; j < VertexSet; j++)
        if ((Slicing[j][tempMatrix[i]] == Slicing[j][tempMatrix[a]])
            && (Slicing[j][tempMatrix[i]] == 1)
            && (tempMatrix[i] != tempMatrix[a]))
            Average[tempMatrix[i]][tempMatrix[a]]++;
}
if (Math.min(SlicingSize[tempMatrix[i]],
        SlicingSize[tempMatrix[a]]) != 0)
        Average[tempMatrix[i]][tempMatrix[a]] = (byte) (100 * 
        (Average[tempMatrix[i]][tempMatrix[a]] / Math
        .min(SlicingSize[tempMatrix[i]],
        SlicingSize[tempMatrix[a]]))); // Evaluate
        
        // Average
        // Average[tempMatrix[i]][tempMatrix[a]] =
        
        100*Average[tempMatrix[i]][tempMatrix[a]]/(SlicingSize[tempMatrix[i]]+SlicingSize[tempMatrix[a]]);
        // System.out.println(11%10);
        sum = (Average[tempMatrix[i]][tempMatrix[a]] + sum);
        // System.out.println(SlicingSize[tempMatrix[i]]);
        // System.out.println(SlicingSize[tempMatrix[a]]);
        //
        System.out.println(Math.min(SlicingSize[tempMatrix[i]],SlicingSize[tempMatrix[a]]));
        //
        System.out.println("Average"+tempMatrix[i]+""+tempMatrix[a]+""="+Average[tempMatrix[i]]
        [tempMatrix[a]]);
        // System.out.println("n");

        }
    }

    // for (int i = 0; i < count3; i++) {
    // for (int j = 0; j < count3; j++) {
    // sum = Average[tempMatrix[i]][tempMatrix[i+j]]+sum;
    // }
    // }
    
    if (count3 - 1 != 0 && count3 != 0)
        AverageValue = sum * 2 / (count3 * (count3 - 1));
        // System.out.println(AverageValue);
        return AverageValue;
    }

public int fitnessFinal(int matrix[], double a, double b) {
    fitnessFinal = (int) (a * FitnessCover(matrix) + b
        * (100 - FitnessOverlapAverage(matrix)));
    return fitnessFinal;
    }
public int Random() {
    Random generator = new Random();
    int random;
    random = generator.nextInt(1000);

    return random;
}

public int evaluate(int indx) {
    int value = 0;
    for (int i = 0; i < VertexSet; i++)
        value = SlicingMatrix[i][indx] + value;
    return value;
}

public void nonempty() {
    for (int i = 0; i < SourceFileLine; i++)
        if (value[i] != 0) {
            for (int j = 0; j < VertexSet; j++)
                Slicing[j][count] = SlicingMatrix[j][i];
            count++;
        }

    System.out.println("count=" + count);
    // System.out.println(count);
}

public void value() {
    for (int i = 0; i < SourceFileLine; i++)
        value[i] = evaluate(i);
}

public void display() {
    for (int i = 0; i < count; i++)
        int number = -1;
        for (int j = 0; j < VertexSet; j++)
            System.out.print("Slicing[" + j + "]" + i + "]" = "
                + Slicing[j][i] + ",
            if (j == 23)
number = Slicing[j][i];

}
System.out.println();
// System.out.println("Slicing[23]=" + number);
System.out.print("Slicing[23][" + i + "]=" + number);
System.out.println();

public void testToken() {
    StringTokenizer token;
    String str = "1 2 3 4";
    token = new StringTokenizer(str);
    while (token.hasMoreTokens()) {
        str = token.nextToken();
        // System.out.println("s1="+str);
    }
    System.out.println("s1="+str);
}