A Genetic Algorithm Test Generator

Susan Khor

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

July, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94745-9
Our file Notre référence
ISBN: 0-612-94745-9

ABSTRACT

A Genetic Algorithm Test Generator

Susan Khor Lay Choo

Use of a genetic algorithm and formal concept analysis to generate test data for branch coverage is explored in a prototype automatic test generator (ATG) called **genet**. **genet** is unique in the sense that it requires minimal source code instrumentation and analysis, and is programming language independent. Besides the novelty of using formal concept analysis within a genetic algorithm, **genet** extends the opportunism of another evolutionary ATG. Experiments were designed to evaluate the effectiveness of **genet** and the importance of selection in the evolution of test data. The results of the experiments indicate **genet** is most effective when selection plays a significant role. This is the case when test solutions for a program are necessarily organized. When it is not necessary for test solutions to resemble each other, adaptation appears to be the more dominant factor and the identification of suitable genetic operators becomes more important. Nevertheless, even in the latter situation, the presence of **genet** accelerated the evolutionary process for our test programs. Notwithstanding equal adaptation instructions, genetics mattered.

Acknowledgements

I am indebted to my supervisor, Dr. Grogono, for believing in my idea for a thesis and taking me on as a student of his half way through my program. The completion of this thesis to its current state is largely the result of his encouragement, patience, resources and high standards.

I set out to do research on genetic algorithms with the notion that I could understand better the question of nature versus nurture. I was and still am not aware if to an educated mind, the one has anything to do with the other. Nevertheless, this thought sustained me.

The production of this document would be highly improbably if not for the following mish mash of things, people, ideas and other matter:

adventure, audacity, Aristotle, Asimov, Austen, aerobics, the Bible, Bach, Beethoven, Greg Butler, basement living, Camellia sinensis, caffeine, chick peas, Chopin, chips, cherries, Copeland, cacao, cinnamon, citeseer, Christmas, Darwin, Diary of a Witch, disappointment, Ecclesiastes, empiricism, mixed emotions, Flora A. DiMaio, fountain pens, French language, Jostein Gaarder, Peter Grogono, Jean Genet, google, How?, how to shower without stretching your arms, Ianina Orenman, International Student Office, ignorance, suspended judgment, knitting, less than perfect, Lamarckism, Mendel, Mom, multi vitamins, music, Halina Monkiewicz, the meditations of Marcus Aurelius, mime, nature, Oprah, opines, expensive writing paper, colour pencils, petunias, Plato, tea pots, quilts, Quebec, J. Rilling, B. Russell, rebel, rewrites, sushi, scones, Sartre, streets of riches, Socrates, summer skies, seven theories of human nature, strawberry flavour, smell of real pencils, tea sets, spice, turmeric, trains, Thoreau, unexpected effect, valerian, why?, winter frosting, words, xerox, yawns, yahoo, zinc.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Testing is a necessary but expensive activity in software production. Therefore it makes economic sense to minimize the cost of software testing. Automatic test generation is seen as key to this cost reduction (Ould 1991).

Test generation is the activity whereby program inputs are crafted to exercise a certain feature of a program as in functional testing or a certain program entity as in structural testing. Such program inputs are test data or tests. Tests for functional testing, also known as black box tests, are derived from the specification of a program while tests for structural testing, also known as white box tests, are based on the implementation of a program. The sets of black box tests and white box tests need not be mutually exclusive. Structural coverage metrics measure the completeness of structural testing.

This thesis explores the use of formal concept analysis and a genetic algorithm to automate generation of white box tests for branch coverage. The system, called **genet**, offers the following advantages over other genetic algorithm based dynamic test generators:

1. A program graph is not required. Elimination of this step lowers **genet**'s adoption barrier and also makes **genet** independent of the choice of programming language.
2. Program instrumentation is simpler.

The results of the experiments we conducted show **genet** to be most effective when test solutions for a program are *necessarily organized*, and the program is written to allow **genet** to learn this organization so that **genet** may exploit common information content to expedite a search by making good selection choices. Organization is a quality measuring the amount of information shared between organisms (Chaitin 1979). This

thesis does a variation on this definition and describes test chromosomes as being necessarily organized when it is essential that specific genes have particular values to execute certain program sub-paths. Tests are not necessarily organized when it is possible for chromosomes with completely different gene values to produce the same behavior. When there is little *necessary* organization amongst test solutions (the organisms in question), we observed that problem appropriate genetic operators (adaptation) played a more significant role than selection. Nevertheless, **genet** could improve upon a pure adaptation search. Our findings with **genet** agree with the theoretical work by Wolpert (1995) and Altenberg (1995).

Chapters 2, 3 and 4 cover pertinent background material on genetic algorithms, test data generation and formal concept analysis. Chapter 5 demonstrates **genet** with an example. Chapter 6 reports on the experiments and discussions leading up to our conclusion. Details of **genet**'s implementation and sample code can be found in Appendix A. Instrumented versions of programs used in the experiments are listed in Appendix B.

Chapter 2 Genetic Algorithms

Genetic algorithms (GAs) simulate biological evolution to search for solutions to problems. In the GA framework defined by Holland (Holland 1975), individuals evolve by means of selection and adaptation to reach their goal. A GA has also been described as a pseudo-random walk through a search space (Burgess 2003). The walk is pseudo-random because a GA is directed in its search through selection pressure and non-deterministic in its adaptation.

GA solutions have been applied to numerous optimization and machine learning problems such as job shop scheduling, classification rule identification and protein structure prediction. Typical components of a GA are (1) representation of a solution, also known as a chromosome or genome, (2) a population of candidate solutions, (3) a fitness function, (4) a selection function and (5) genetic operators to perform adaptation.

2.1 Representation

Chromosomes are typically fixed in size and encoded in binary, Gray (1953), integer or floating point. Non-standard representations include data structures like lists, multidimensional arrays, trees and matrices. A chromosome may be variable in size.

Representation can influence the success of a GA. Table 2.1 shows the binary and Gray code equivalents for eight integers. In Gray code, a string N differs from its adjacent strings N-1 and N+1 in exactly one position. To see how representation influences a GA search, consider adapting the binary and Gray representations in Table 2.1 by changing the leftmost bit. In binary encoding, integer 3 becomes integer 7 and integer 0 becomes integer 4. In Gray encoding, integer 3 becomes integer 4 and integer 0

becomes integer 7. Depending on the problem at hand, this adaptation may or may not help the GA to converge to an optimal solution. Single random bit change is a common form of mutation in GA.

Table 2.1 Gray code example

Integer	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Deciding on which representation to use depends on the problem being solved. Radcliffe (1995) found: "If no domain-specific knowledge is used in selecting an appropriate representation, the algorithm will have no opportunity to exceed the performance of an enumerative search." Mitchell (1996, page 158) refers to Lawrence Davis, a seasoned designer of real world GA solutions, who "...strongly advocates using whatever encoding is the most natural for [the] problem".

2.2 Population

A GA works by evolving a set of initial guesses into an increasingly fit population of candidate solutions. The set of initial chromosomes may be created manually, loaded from a previous run or generated at random. Seeding the initial population with good guesses may increase the likelihood that an optimal solution is found quickly. However this requires knowing what constitutes a good guess.

Depending on the replacement strategy, subsequent populations may include chromosomes from more than one generation. In Goldberg's (1989) *simple genetic*

algorithm, the current generation completely replaces the previous generation. In *steady-state* GA, only a portion of the chromosomes from the previous generation is replaced by that of the current generation. A *deme* GA (Cantu-Paz 1997) maintains multiple sub-populations in parallel.

Deciding on the population size is not a straightforward matter. Experiments by De Jong (1975) suggest a population size of 50 – 100. A large population increases the number and possibly the spread of points sampled in the search space so that the GA may be less likely to get caught in a local optimum. Evaluating a large population can be expensive. A small population is cheaper to evaluate but is less likely to be sufficiently diverse. As a consequence, a GA with a small population will converge prematurely, according to De Jong. A GA is said to converge prematurely when it no longer has the ability to produce fitter chromosomes. Experiments by Grefenstette (1986) indicate otherwise – that a small population performs better than a larger one. Syswerda (1991) advices: "...the most effective population size is dependent on the problem being solved, the representation used and the operators manipulating the representation".

2.3 Fitness function

A fitness function measures the suitability of a chromosome to meet a GA's objective. For example, if the objective of a GA is the integer value "63" (or "111 111" in binary) and binary encoding is used, then a chromosome "111 010" is fitter than another chromosome "001 001". Effective fitness functions are problem specific.

One of the difficulties with GA is finding a suitable fitness function that expresses the problem well enough so that every point in the search space can be assigned a fitness value that is informative enough for the GA to discriminate chromosomes at a useful

level of detail. For example, a fitness function that can only tell if a chromosome is fit or unfit, is unlikely to be very useful. More useful would be a fitness function that can tell which of the unfit chromosomes is closer to a solution. Moreover, as fitness functions are evaluated repeatedly, they should be easy to compute.

2.4 Selection function

Chromosomes are usually selected for adaptation on the basis of their fitness values relative to other individuals in the population and to the environment at hand. Three classic selection strategies are *Roulette-Wheel* (Goldberg 1989), *Rank* (Goldberg 1991) and *Tournament* (Grefenstette 1989). The following example illustrates these three strategies: Let the fitness values for a population of three chromosomes i, j and k be 3, 16 and 1, with j being the fittest individual.

Roulette-Wheel (RW) is a fitness-proportionate strategy. This means the selection probability of a chromosome is proportional to its fitness value divided by the population's total fitness value. The total fitness value of the population in our example is 20 (3+16+1). Chromosome i occupies 15% of the roulette wheel (Table 2.2). When the wheel is spun, the probability that the roulette lands on the space occupied by chromosome i is 0.15.

Table 2.2 Roulette Wheel and Rank selection

Chromosome	Fitness	RW probability	Rank	R probability	
i	3	0.15	2	0.5	0.33
j	16	0.80	1	0.9	0.60
k	1	0.05	3	0.1	0.07
	20	1.00		1.5	1.00

The RW strategy favors fitter chromosomes. One consequence of this is that when fitness variance is high, super fit chromosomes are likely to have a large number of

offspring and saturate the population with their genes. Depending on the nature of the problem being solved, such high selection pressure may reduce a population's diversity to a point where a GA converges prematurely. Lowering selection pressure to slow down the convergence process may lead to better solutions, but this effect is not guaranteed.

Rank and Tournament selection are non-fitness proportionate strategies. A Rank strategy sorts chromosomes according to their fitness values and assigns a selection probability based on their rank order. The rank values in our example population are 2, 1 and 3 for chromosomes *i*, *j* and *k* respectively (Table 2.2). If the selection probability function is $(-0.4r + 1.3)$ (Burgess 2003) where r is a rank value, then the selection probability for chromosomes *i*, *j*, and *k* are 0.5, 0.9 and 0.1 respectively.

Ranking reduces selection pressure when the fitness variance is high. When these probabilities are scaled to make the total probability equal to 1.0, the probability of chromosome *j* being selected under a Rank strategy is 0.6 which is less than its probability of 0.8 under RW. Conversely, the selection probability of chromosomes *i* and *k* are higher under Rank than with RW.

The selection pressure exerted by a Tournament strategy is quite similar to Rank (Mitchell 1996, page 171). However Tournament is considered more efficient of the two because it does not sort fitness values. Under a Tournament strategy, a group of two or more chromosomes are first chosen randomly from the population. Then the chromosome with the highest fitness value from this group is selected and the remaining chromosomes in the group are returned to the population. This two step process continues until the required number of chromosomes is selected. The risk with a strategy like Tournament is that fitter individuals may not get selected at all, and depending on the population

strategy used, fit individuals may be lost. One way to counter this risk is to use *elitism* (De Jong 1975) where a certain number of the fittest chromosomes are retained in every generation.

2.5 Genetic operators

Two common genetic operators are recombination, also known as crossover, and mutation. Recombination is the random exchange of genes between chromosomes. Mutation is a random change within a chromosome. A seldom used genetic operator is inversion. It reverses the order of genes in a section of a chromosome. For example, an inversion of chromosome 11.001.01 at positions 2 and 5 (indicated by the dots) becomes 11 100 01.

Table 2.3 Illustration of crossover operators

Parent	One point (at 3)	Two point (at 2 and 4)	Uniform (at 1)
101001	101 110	10 01 01	1 11100
010110	010 001	01 10 10	0 00011

Simple forms of crossover are one-point, two-point and uniform. Table 2.3 illustrates the effects of these crossover operators on a pair of one-dimensional chromosomes. One-point crossover exchanges the segments between a randomly chosen loci and the tail end of the chromosome pair. Two-point crossover exchanges the segments between two randomly chosen loci of the chromosome pair. Uniform crossover exchanges genes at alternate positions starting from either the first or second loci. Uniform[1] crossover outperformed one- and two- point crossover, and two-point crossover outperformed one-point in the experiments done by Syswerda (1989). A GA's crossover

[1] Syswerda's definition of uniform crossover is more general than the one used in this thesis. Essentially our mask is fixed – alternate bits are exchanged.

rate determines the size of its mating pool (Haupt 1989, page 106). For example, a GA with a population size of 100 and a crossover rate of 0.60 will have 60 chromosomes selected as parents. The meaning of crossover rate in **genet** is slightly different from this definition. In **genet**, the crossover rate defines the number of times the crossover operator is applied to the parent pool. Each crossover in **genet** produces two offspring.

Mutation produces a single offspring from a single parent chromosome. This is typically accomplished by randomly changing the value of a gene in a parent. Mutation helps to introduce new gene values into the population and is necessary to maintain variation in the gene pool. Without mutation, the genetic material of individuals is limited to the initial population. A GA's mutation rate quantifies how much of a chromosome can expect to be changed by a mutation (Haupt 1989, page 106). The effect of a mutation rate depends on representation. In **genet**, the mutation rate defines the number of times the mutation operator is applied to the parent pool. Each mutation in **genet** produces a single offspring.

Mutation encourages the GA to search new regions in the search space but may inadvertently destroy fit sections of chromosomes (good *building blocks*). In fact neither crossover nor mutation guarantees that good blocks of genes will be preserved in subsequent generations. According to Holland (1975), his simple genetic algorithm works because fit building blocks or schema are identified and combined to make larger fitter building blocks. A GA searches by exploring the search space for promising regions. Once a promising region is detected, it is advantageous for the GA to stay within the region so that it can fully exploit it. Mutation is the main operator for exploration (global search) while recombination is the main operator for exploitation (local search) (Burgess

2003). If a GA's mutation rate is too low or its recombination rate is too high for a problem, the GA might be stuck in a sub-optimal region. If its mutation rate is too high or its recombination rate too low, the GA might not stay long enough in a region to exploit it. How then should crossover and mutation rates be set?

In general, a GA's crossover rate tends to be much higher than its mutation rate. De Jong (1975) suggests crossover rates should be about 0.60 and mutation rates around 0.001. Haupt (1989, page 106) concludes that a GA is sensitive to its mutation rate and suggests a higher mutation rate, between 0.1 and 0.4. Grefenstette (1989) found that "...very good performance can be obtained with a range of GA control parameter settings." Due to the problem sensitive nature of a GA's representation, genetic operators and parameter settings, there are GAs which adapt their parameters (Mitchell 1996, page 177).

2.6 Basic design

The basic design of a genetic algorithm is:

1. use fitness function to evaluate chromosomes in generation t
2. if solution found or stopping criterion is reached, exit
3. use the selection function to choose parents
4. apply crossover and mutation operators on parents to make generation t+1
5. go to step 1

Defining the stopping criterion for a GA is yet another decision a designer needs to make. Possible criterion include stopping when a GA has met its objective, when a GA has not improved for a number of generations and when a GA has reached a certain number of generations.

Chapter 3 Dynamic Test Generators and Their Evaluation

3.1 Overview of dynamic test generators

Dynamic test data generators execute programs to produce test input. The alternative is static test data generators (Clarke 1976) which do not execute the program under test. Symbolic execution is the common static method used to calculate the constraints on input variables. Dynamic test generation methods have several advantages: they can handle loops, arrays, pointers, functions and other dynamic constructs, more easily than static methods. However, dynamic methods can be more expensive than static methods since they involve multiple executions of a program under test.

Typical components of a systematic dynamic test generator are: (1) an executable program with its source instrumented per requirements of the test generator, (2) a testing criterion, (3) a test generation method, and (4) a search mechanism to guide the test generation. A random test generator is a dynamic test generator and has components (1), (2) and (3). A random test generator is not a systematic dynamic test generator since it does not have a specific search mechanism.

The testing criterion measures the progress made by a dynamic test generator. Some examples of structural testing criteria include statement coverage (Pargas 1999), branch coverage (Jones 1996) and condition-decision coverage (McGraw 1998). Complete statement coverage is the situation where every statement in a program is exercised by at least one test. If every branch in a program is triggered by at least one test in a set of tests, then the test set provides full branch coverage. A predicate or decision may consist of multiple conditions connected by logical operators. To achieve full condition-decision coverage, a test set must achieve full branch coverage and each

condition within a predicate (decision) must evaluate to true at least once and to false at least once. This is a stricter testing criterion than branch coverage. Tracey (1998) used a functional testing criterion using preconditions and negated post conditions to detect flawed functionality.

Search mechanisms which have been employed by systematic dynamic test generators include directed search (Korel 1990), genetic algorithms (Jones 1996; McGraw1998 and Pargas 1999), simulated annealing (Tracey 1998) and tabu search (Tracey 1997). In simulated annealing, better candidate solutions are always accepted while worse candidate solutions are accepted with a decreasing probability over the simulation. In tabu search, candidate solutions are found based on the history of the search move sequence and whether a move is considered, by some heuristic, as tabu or forbidden. A move may be classified as tabu to prevent recycling of candidate solutions. However, to obtain the best solution, it is also possible to forget that a move is tabu.

3.2 The first systematic dynamic test generator

The first systematic dynamic test generator was TestGen (Korel 1990). In TestGen, a control flow graph for a program under test is constructed. Each branch in the control flow graph is labeled with a branch predicate that describes the conditions under which the branch will execute. For example, if a true branch from a predicate node is $x > 5$ than the false branch must[2] be $x \leq 5$. The program is instrumented so that every branch has a branch function. In the case of branch predicate ($x > 5$), the branch function would be $f(x) = 5 - x$; and for ($x \leq 5$), $f(x) = x - 5$. Figure 3.1 illustrates this point.

[2] Assumes an if statement jumps to two different positions. The arithmetic conditional statement in early version of Fortran jumps to three positions depending on whether the arithmetic expression is less than zero, equal to zero or greater than zero.

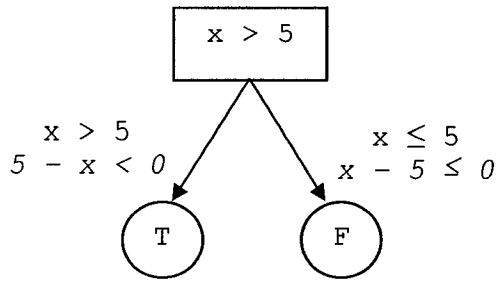


Figure 3.1 Branch function example

Table 3.1 contains the branch functions for all relational operators. To traverse either branch, the branch function cannot be positive. For example, the true branch in Figure 3.1 will execute only if the branch function (5 – x) is less than 0 and this will only happen if x > 5. Similarly, the false branch will execute only if the branch function (x – 5) is less than or equal to 0 and this will only happen if x ≤ 5.

Table 3.1 Branch functions

Branch Predicate	Branch Function	Relational Operator
E1 > E2	E2 – E1	>
E1 ≥ E2	E2 – E1	≥
E1 < E2	E1 – E2	<
E1 ≤ E2	E1 – E2	≤
E1 = E2	Abs(E1 – E2)	=
E1 <> E2	Abs(E1 – E2)	<>

The first version of TestGen is a path-oriented dynamic test generator because a path is pre-selected from the control flow graph as the target path and TestGen tries to find test data to execute the target path. A non-trivial path in a program consists of a sequence of branch predicates. The branch functions associated with a sequence of branch predicates are used to guide the search for test data that will cause a target path to execute. Each time an execution diverges from a target path, input variables are adjusted one at a time (with other variables held constant) and the value of the branch function where the divergence occurred (the sub-goal branch function value) is monitored to learn

whether the variable affects the sub-goal branch function and if so the direction in which the adjustment should occur, i.e. should the value of the variable be increased or decreased to reduce the sub-goal branch function value. Korel (1990) named this the *exploratory search* phase.

If an increase in a variable in the exploratory search phase reduces the sub-goal branch function value, then in the *pattern search* phase, the variable is increased by successively larger steps as long as the value of the sub-goal branch function continues to decrease. If a pattern move causes a constraint violation (the new variable value causes the execution to diverge from the path leading to the sub-goal branch), the step size is reduced until a successful pattern move can be made. If the sub-goal branch function value starts to increase, another exploratory search is conducted and this process repeats itself until the sub-goal branch function value becomes negative (or zero in some cases). The search may also terminate if the exploratory search fails to find any variable that will decrease the sub-goal branch function value. It is important to note that TestGen will only accept a move if the move reduces the sub-goal branch function value without causing any constraint violation.

To illustrate the exploratory and pattern searches in action, consider the control flow tree in Figure 3.2. Assume an integer domain. The target path is 1 → 2 → 5. The current value of x is -5 which causes the path 1 → 2 → 4 to execute. So, the divergence occurs at node 2, and the branch function to minimize f(x), is $x^2 - 4$. The current value of this branch function f(-5), is 21. An exploratory search commences with a -1 step applied to x, giving f(-6) = 32, which is an increase. Therefore x is incremented by one, giving

f(-4) = 12, which is a decrease. The exploratory search has successfully identified an influencing variable and a direction for the pattern search.

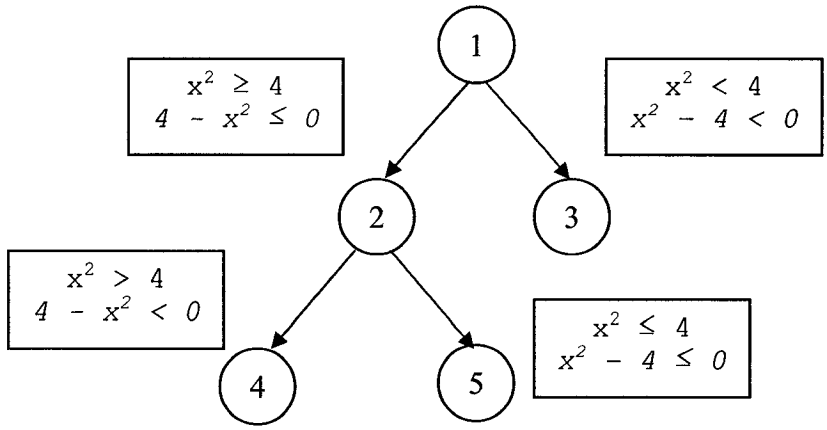


Figure 3.2 Sample control flow tree

Table 3.2 Search moves for connected domain

	Step	x + step	Constraint		Sub-goal branch function		Executed path
			$4 - x^2 \leq 0$		$x^2 - 4 \leq 0$		
Current	0	-5	-21	T	21	F	1 → 2 → 4
Exploratory	-1	-6			32	F	
	+1	-4			12	F	
Pattern	+2	-3			5	F	
	+4	-1	3	F	-3	T	
	+3	-2	0	T	0	T	1 → 2 → 5

In the pattern search phase, a larger positive step is taken, +2, giving f(-3) = 5 which is a further decrease to the sub-goal branch function. Next an even larger positive step is taken, +4, giving f(-1) = -3 which satisfies our sub-goal branch function $x^2 - 4 \leq 0$, but violates an earlier constraint of $4 - x^2 \leq 0$. x = -1 would traverse path 1 → 3 if the program was run. So this pattern move is not acceptable and a smaller step, +3, is taken. Now x = -2 and f(-2) = 0 satisfies our sub-goal without violating an earlier constraint. Therefore x = -2 is accepted as a solution to our sub-goal and this solution happens to execute our target path 1 → 2 → 5. Table 3.2 summarizes the sequence of moves described here.

One difficulty with the directed search described here is that some input variables may not have any effect on a branch function, and when there are many input variables, much effort could be spent needlessly exploring to find a variable that affects a branch function. To improve efficiency, Korel (1990) suggested using data flow analysis to identify input variables that influence the value of a branch function.

Another source of inefficiency is infeasible paths. A path is infeasible if there are no program inputs that can satisfy its constraints, i.e. there is no test data within the problem domain that will cause the path to execute. The program path 1 → 4 → 5 in Figure 3.3 is infeasible. Since it is not possible, in general, to identify infeasible paths; a lot of effort is wasted if a target path turns out to be actually infeasible. Because of these disadvantages, TestGen later became a goal-oriented test generator, with a *chaining* approach (Ferguson 1996).

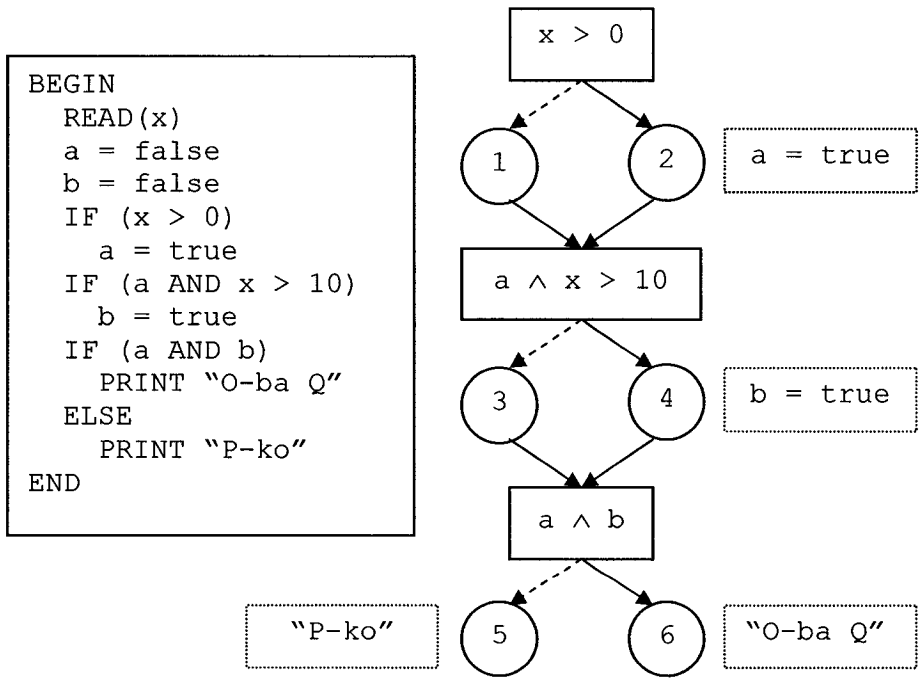


Figure 3.3 Sample program and control flow tree

A goal-oriented test generator identifies a target node but does not specify the program path that a test should take to reach the target node. Suppose node 3 is the target node in Figure 3.3. Then any test that exercises node 3, either by way of node 1 or node 2, is accepted by a goal-oriented test generator as a solution. In this case, any x value less than or equal to 10 is a solution. Under a path-oriented test generator, a program path to node 3 would have been pre-determined and the path-oriented test generator would have had to follow this path. If the pre-determined path were 1 → 3, then only x values less than or equal to 0 is a solution. Goal orientation frees TestGen from the infeasible path problem, but not unreachable statements or branches.

Chaining (Ferguson 1996) was introduced to give some guidance to the goal-oriented approach without pre-determining the complete execution path. It uses data dependency analysis to identify nodes that must be executed before the target node. Doing this significantly improves the chances of finding test data to exercise the target node (Ferguson 1996). Suppose the test generator has so far failed to alter the execution flow to the target node, node 6 in Figure 3.3. Chaining analyzes that the predicate controlling node 6 uses variables a and b, and that the last definitions of these variables occur at nodes 2 and 4. Thus, the chaining approach concludes that nodes 2 and 4 should be executed prior to node 6 and the test generator is well advised to find tests solutions for nodes 2 and 4 before attempting node 6.

3.3 Why genetic algorithms?

One reason frequently given for using genetic algorithms is their ability to escape from "difficult areas" in a search space. The search space for a problem is defined as the set of all possible candidate solutions for the problem. Consider a problem with four variables

and an integer value range of [1, 3] for each variable. Five points or candidate solutions in this search space are (1, 2, 3, 2), (2, 1, 3, 3), (3, 2, 3, 3), (1, 1, 1, 2) and (2, 2, 2, 2). This search space has a size of 3^4 and a dimensionality of four.

Imagine a search space as a multidimensional landscape with many peaks and valleys. Two "difficult areas" in such a landscape are local minima or local maxima and plateau. A local minimum is a valley that is lower than its surrounding area but higher than the lowest valley in the search space. Similarly, a local maximum is a hill that is higher than its surrounding area but lower than the highest peak in the search space. A plateau is a flat area in the search space with no nearby hills or valleys. A "difficult area" is difficult from the perspective of hill climbing search algorithms. Hill climbing performs a local search and only accepts a candidate that improves the current solution. Hence, a hill climbing algorithm will refuse to descend from a local maximum, and might not be able to find its way off a plateau.

GAs can escape from these "difficult areas" because they evaluate more than one candidate solution at a time and they accept less fit chromosomes which may be produced as a result of selection and adaptation. The directed search method used in TestGen is similar to hill climbing and therefore is also susceptible to the local minima problem. Korel (1990) acknowledged that local minima can prevent sub-goals from being solved.

To demonstrate how this can happen, suppose now that the domain for x in Figure 3.2 is all integers in $\{x \mid x < -2 \text{ or } x \geq 0\}$. This domain is disconnected. We try to use the same search steps as in Table 3.2, but find that we are unable to do a +4 step because -1 is not in the domain of x. So we take a bigger step, +5. This causes a constraint violation for an earlier predicate, and according to TestGen's pattern search rules, the step size should

now be reduced until a successful pattern move can be made. But any such reduction in step size will either land x outside its domain or increase the sub-goal branch function. Therefore, in this situation, the search failed to find a solution for the path 1 → 2 → 5. Table 3.3 summarizes the steps.

Table 3.3 Search moves for disconnected domain

	Step	x + Step	Constraint		Sub-goal branch function		Executed path
			$4 - x^2 \leq 0$		$x^2 - 4 \leq 0$		
Current		-5	-21	T	21	F	1 → 2 → 4
Exploratory	-1	-6			32	F	
	+1	-4			12	F	
Pattern	+2	-3			5	F	
	+5	0	4	F	-4	T	

If we took a larger step, say +6, this would increase the sub-goal branch function to -3, and TestGen would reject such a move even though continuing the search in this direction would have lead TestGen to find a solution at x = 2. If there were more variables involved, TestGen would start another exploratory search to find a search direction for another variable. However, since x is the only variable, TestGen thinks no further progress can be made and so it stops searching.

An obvious advantage GA has over hill climbing techniques for test data generation is parallelism. The search in GA progresses from multiple search points and this is appropriate if we consider test data generation as a constraint satisfaction problem (CSP). A CSP seeks *any* solution that satisfies all of its constraint conditions. There may be more than one point in the search space that satisfies a CSP and it does not matter which point the GA finds. In contrast, an optimization problem seeks the best solution.

To put this into a test data generation context, the search space for a program is its input space. A feasible program path is defined by a set of conditions that when satisfied,

causes the path to execute. In other words, a feasible program path is a solvable CSP and all points in the program's input space that is a solution to a feasible program path forms an equivalence class[3]. We call such an equivalence class of points in the input space a *sub-domain*. Naturally there can be one or more sub-domains per program and some sub-domains may have more points than others. The cardinality or size of a sub-domain refers to the number of points in it. There may be large variations in how points of a sub-domain are distributed, i.e. they may be widely scattered across the input space or they may cluster at one corner of the input space. When points in a sub-domain are dispersed over the input space, the sub-domain is said to be disconnected.

The task of a test data generator that has full structural coverage as its objective is to generate at least one test from every sub-domain of a program's input space. GA, with its ability to do adaptive search in parallel is an appropriate search mechanism for this task. The three dynamic test generators that we describe next use GA as their search mechanism.

3.4 GA-based dynamic test generators

3.4.1 Domain testing

The objective of this goal-oriented dynamic GA test generator (Jones 1996) is to find test data situated as close as possible to sub-domain boundaries where predicate values are likely to switch from true to false and vice versa. Branch coverage criterion with provisions for loops (explained below) is used and branch node targets are decided by a breadth-first sweep of the control flow tree.

[3] Programs are treated as deterministic functions. So it is not possible for there to be more than one outcome for a test given the same set of conditions, e.g. program state.

To use this dynamic test generator, every branch of the program under test is instrumented with two procedures: (1) CHECK_BRANCH, to register that a node has been visited; and (2) LOOKING_BRANCH, to determine the fitness of a test, which is set differently depending on the current target node. If a test exercises the current target node, then the LOOKING_BRANCH procedure at the current target node will calculate a high fitness value for the test. If a test exercises the sibling node of the current target node, then the fitness of that test is calculated using Hamming distance, which measures the number of positions in a test chromosome where the bit value is different.

In case of loops, two other monitoring procedures are used: CHECK_BRANCH_LOOP and LOOKING_BRANCH_LOOP. Loops are unrolled one, two or more times, and the fitness of tests exercising loops "is related to the difference between the actual and required number of iterations" (Jones 1996).

The chromosome for this test generator is a concatenation of all input variables into a single bit string. The initial population is randomly formed. Its size is usually equal the length of the chromosome bit string. This GA was "trained" on a quadratic equation solver to find the most suitable type of and probability for crossover and mutation. Jones (1996) found that uniform crossover with a crossover probability around 0.5, and mutation with a probability that is reciprocal of the bit string length and "a weighted mutation of the five least significant bits" were most suitable. These parameter values are used in other experiments on this GA. Population diversity is maintained with random parent selection and with a hybrid of elite survival and random selection chromosome replacement strategy. A limit of 100 to 2000 generations is set.

The number of tests required to achieve full coverage by this dynamic GA test generator and by random testing were compared. The results are summarized in Table 3.4 and it shows GA testing requires fewer tests than random testing to achieve the testing criterion. The GA test generator was run on linear and binary searches, and generic quick-sort to demonstrate that test could be complex data structures, but no empirical data were reported.

Table 3.4 Number of tests required for full coverage.

	GA	Random
Quadratic equation solver	1200	7400
Triangle classifier	18000	163000
Remainder calculation	900	63000

From their experiments, Jones (1996) concludes that genetic algorithms really prove their worth on non-linear predicates and are powerful for locating test from disconnected sub-domains with very small cardinalities. Non-linearity refers to a system, model or equation where the outcome does not change proportionally to changes in the values of its input variables. Sthamer (1995) reached a similar conclusion about the efficacy of GA for test data generation. In his dissertation, Sthamer says that GA testing requires fewer tests than random testing when the density of solutions is quite low. Low solution density has a direct relationship with small disconnected sub-domains.

3.4.2 GADGET (Genetic Algorithm Data GEneration Tool)

The testing criterion for GADGET (McGraw 1998) is condition-decision coverage. GADGET is a goal-oriented dynamic GA test generator and uses a genetic algorithm to do function minimization of branch functions (Korel 1990). A decision table (Chang 1996) is used to help GADGET select target branches. GADGET targets one branch at a

time. Serendipitous branch coverage is allowed, i.e. fortuitous coverage of non-target branches.

Table 3.5 is a decision table where decisions 3 and 4 are partially covered and decision 5 is completely uncovered. The strategy is to choose uncovered branches of partially covered decisions over branches of completely uncovered decisions. The logic here is that GADGET has already learnt how to reach the covered branches of partially covered decisions and therefore the uncovered branches of partially covered decisions have a better chance of being covered than those branches of completely uncovered decisions which GADGET does not know how to reach yet. To illustrate this, GADGET will choose either the false branch of decision 3 or the true branch of decision 4 as the next target branch, over the branches of decision 5.

Table 3.5 Decision Table

Decision	True branch	False branch
1	X	X
2	X	X
3	X	
4		X
5		

McGraw and his colleagues experimented with two genetic algorithms: a standard GA and a differential GA. We focus on the standard GA version; the differential GA is more suited for numerical minimization problems. A test solution is represented to the GA as a bit string. One point crossover at a random bit position is used. Mutation is applied, with a low mutation rate. Parents are selected using a roulette-wheel selection scheme and an individual can be chosen only once. Of the four individuals (two parents and their two offspring), two most fit are kept in the next generation, unless they happen to be identical in which case an individual is created at random and added to the next

generation. Adaptation of a previous generation continues until the new generation is the same size as the previous one. The population size is kept steady from generation to generation. The evaluation-selection-adaptation cycle is repeated until either a target is covered or a termination condition is reached.

The performance of GADGET against random testing on a number of small programs was reported and is reproduced in Table 3.6. In every case, at least one of the GA performed better than or as well as random testing. The Triangle classification program shows the largest performance difference.

Table 3.6 Highest coverage percentage by each method during a series of five runs

Program	random	GA	Differential-GA
Binary search	80	70	100
Bubble sort 1	100	100	100
Bubble sort 2	100	100	100
Number of days between two dates	87.5	100	100
Euclidean greatest common denominator	100	100	100
Insertion sort	100	92.3	100
Computing the median	100	100	100
Quadratic formula	75	75	75
Warshall's algorithm	91.7	100	100
Triangle classification	48.6	94.29	84.3

However it is not clear if this performance difference is due to the efforts of the GA or the weakness of random testing. Random testing is sensitive to the interval from which tests are chosen from (DeMillo 1978). The sample tests given in (McGraw 1998) have Triangle input integers ranging from -1,802,686,561 to 1,961,702,355. Note that any integer combination with at least one negative value immediately invalidates the combination as any kind of triangle. This constraint dramatically reduces the probability that a random combination from the test interval is a triangle, and creates an input space that is difficult for random testing; but conducive for GA testing. Recall from the previous section that GA testing performs well when the density of solutions is low.

Having half of all points in the input space invalid certainly helps to decrease the solution density.

When GADGET was tested on a medium-sized program, part of a *b737* autopilot system which has 69 decision points; GADGET achieved more than 93% condition-decision coverage while random managed 55% coverage. This result follows the trend McGraw (1998) recognized when reviewing previous experimental results on test generation (Chang 1996 and Ferguson 1996). This trend is that random test generation can be at least as effective as heuristic based test generation on small, simple programs and with less demanding test coverage criteria. Simple branch coverage is considered one such less demanding criterion. The test programs in Table 3.6 have an average of 30 lines of code and simple decisions.

One of the reasons McGraw (1998) gave for worse performance by random testing on larger and more complicated programs is the lack of a mechanism to set the stage for "coincidental discovery of test inputs", which is a phenomenon that occurs more commonly on larger programs. GADGET is more successful at "coincidental discovery" on large programs because its GA acts as such a mechanism.

Further experiments by McGraw (1998) indicate GADGET performed better compared to random test generation the more complex programs got with respect to nesting factor and conditional factor. The nesting factor describes how deeply predicates are nested and the conditional factor relates to the number of Boolean conditions in each decision.

Most of the decisions that GADGET failed to cover contained Boolean variables or enumerated types. "IF (PGBLK)" is an example of a decision with a Boolean variable.

GADGET's minimization function at this branch can evaluate to two values only, one signifying true and the other false. From such a two valued fitness function, any GA would be unable to tell if one test candidate is "less false" than another. This is one limitation of GADGET. There were also decisions without Boolean variables but with multiple conditions, that GADGET had difficulty covering. McGraw (1998) explains that the variables in these decisions may be complicated functions of test data. Therefore the actual complexity of a decision may not be obvious from its syntax.

3.4.3 TGen

TGen (Pargas 1999) uses a control dependence graph to evaluate the fitness of a test solution. A control dependence graph (CDG) is composed of nodes representing statements and directed edges denoting control dependency of the sink node on the source node. A node Z is control dependent on another node X if there is a path $\langle X, Y_1, ..., Y_n, Z\rangle$ in the control flow graph such that Z post-dominates all nodes in the sub-path $\langle Y_1, ..., Y_n\rangle$ and Z does not post-dominate X. Node B post-dominates node A if all directed paths on the control flow graph from A to the exit node must pass through B.

An acyclic path from the root to a target node of a CDG is called a *control-dependence predicate path* (Pargas 1999). During its initialization, TGen generates the control-dependence predicate paths from the CDG of the program under test and the list of target nodes provided to it. A control-dependence predicate path contains a set of predicates that when satisfied by a test solution causes the statements associated with the target node to execute. TGen runs the program under test with every new test candidate and records the predicates which get executed by every test. The fitness of a test candidate for the current target node depends on the number of predicates it managed to

cover on the control-dependence predicate path for the current target node. More predicate matches mean higher fitness.

A chromosome in TGen is a concatenation of actual values of the program inputs. One point crossover is used with a probability of 0.9 and mutation at 0.1. Roulette-wheel parent selection is practiced, and a chromosome may be selected multiple times. The number of parent chromosomes equals the size of the new generation and the existing generation is completely replaced by its offspring.

Experiments comparing TGen and random testing were performed on 6 C programs. The testing criterion for all programs was statement coverage; but branch coverage was also required for the *Tritype* program. The population size for the experiments is 100. Table 3.7 summarizes the results of the experiments. In the case of *Bub*, *Find* and *Mid*, Pargas (1999) reported that both TGen and Random achieved full statement coverage almost immediately, i.e. in the initial population. For the other three programs, *Bisect*, *Fourballs* and *Tritype*, TGen required fewer generations, on average, than Random to reach full statement coverage.

Table 3.7 is interesting for two reasons. First, if we take the lines of code and cyclomatic complexity[4] metrics as measures of program size and complexity, and if we follow the line of reasoning from the previous section that it is harder to cover programs of greater size and complexity (therefore a GA search is needed); then the following comparisons are surprising: (1) that *Bisect* is more difficult to cover than *Bub* and (2) that *Find* is just as easy to cover as *Mid*. Second, if we follow the line of reasoning from the previous section that it is harder to satisfy more demanding test criterion for programs of

[4] Cyclomatic complexity measures the number of linearly independent program paths. It was introduced by Thomas McCabe in 1976 and is one of the more widely accepted static software metric.

similar size and complexity, then the data for *Tritype* is surprising because both TGen and Random took fewer generations, on average, to satisfy branch coverage than to satisfy a less demanding criterion, statement coverage.

Table 3.7 Average number of generations to reach 100% coverage from 32 runs

Program	Lines of Code	Cyclomatic Complexity	TGen Mean	Random Mean
Bub.c	32	4	1	1
Find.c	66	5	1	1
Mid.c	21	4	1	1
Bisect.c	36	3	29	40
Fourballs.c	82	7	95	159
Tritype.c	61	7	145	1259
Tritype.c (branch)			132	1100

Pargas (1999) concludes that TGen performs better than Random when the program code contains nested conditionals or nested loops with difficult to satisfy predicates.

3.5 Random test generation

Random testing selects points at random from a program's input space. This method is considered to be the cheapest and easiest. Hence systematic test generators should at least outperform random testing to justify their additional cost (Ince 1987). However, the volume of tests that random test is likely to generate to meet a test criterion is a major disadvantage of this method since checking test results, which involves knowing what the expected test results are, is another costly and time consuming process.

Random testing does not use any kind of mechanism to guide its test generation. Nor does it adapt previously generated test to produce new tests. In its basic form, every point in an input space has an equal chance under random testing of being selected as the next test point, and selection of a test point is independent of previous selections. Because

of this arbitrariness, random testing is perceived as an ineffective testing method. Further, DeMillo (1978) reports that "the adequacy of random test data is very dependent on the interval from which the data is drawn (i.e., problem-specific information is needed to obtain good results)".

However, when compared with simple partition testing, Duran (1981) found random testing can be more cost effective at finding faults. Partition testing is a testing approach that divides the input space of a program into subsets and selects at least one test from each subset of the partition. Path testing is a kind of partition testing.

3.6 Other related work

Harman (2002) found the performance of the *DaimlerChrysler Evolutionary Testing System* could be improved by using program transformations to remove Boolean or flag variables. This evolutionary testing system, like GADGET, uses a numeric fitness function. A program transformation is a function applied to a program to produce a semantically equivalent, but syntactically different program. Harman (2002) describes the effect of removing flag variables from a program as changing the search space character for a meta-heuristic search method like GA, from "a coarse fitness landscape with a single super-fit plateau and a single super-unfit plateau (corresponding to the two possible values of the flag variable)" to "a smoother fitness landscape" capable of providing more fitness information to the search method.

Non-arithmetic fitness function like the one used in TGen avoids the flag variable problem to a certain extent because it is at least able to distinguish between test candidates that come close to the target node and those that do not. Take a hypothetical control flow tree in Figure 3.4 as an example. The target is node 5. A non-arithmetic

fitness function based on proximity will assign a higher fitness value to T1 than T2 because T1 executes more of the nodes that a test for node 5 must cover. An arithmetic fitness function will assign the same fitness value to T1 and T2. However, the non-arithmetic fitness function based on proximity does *not* distinguish between the fitness values of T1 and T3.

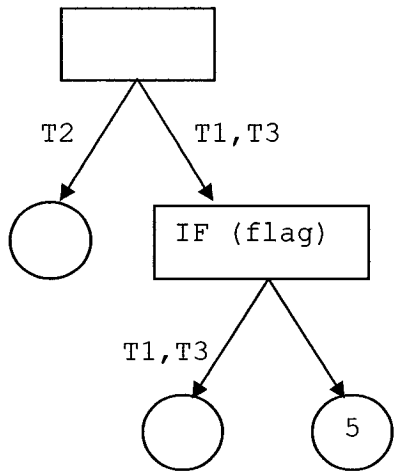


Figure 3.4 A hypothetical control flow tree

3.7 Summary

The main points to note from this chapter are:

1. Test data generation for an arbitrary program may be viewed as a non-linear constraint satisfaction problem. Given the nature of this problem, the use of genetic algorithms to generate test data is justified, and we demonstrated this with an example (Figure 3.2).
2. Test data generation using genetic algorithms (evolutionary testing) has been proven on "real world" programs.
3. Several limitations of previous approaches are:
    a. Complicated instrumentation of predicates

b. Reliance on availability of a program graph

c. In some cases, inability to handle Boolean variables and non-arithmetic conditions

4. All test generators surveyed compared themselves with random testing.

5. Results of these experiments reveal certain qualitative factors which make GA testing more efficient at structural coverage than random testing. These factors are:

   a. Large program size.

   b. Non-linear predicates.

   c. High structural complexity. Structural complexity includes the number of decisions points in the program under test, the lengths of program paths, the number of conditions in a decision, and the nesting level of a decision.

   d. Demanding testing criterion.

   e. Small, disconnected sub-domains.

   f. Low solution density. One of the weaknesses of random testing is that it is sensitive to the feasibility of the input space. If the input space is defined in such a way that it is highly feasible to find test solutions, i.e. the solution density is high; then random testing has a better chance of stumbling upon test solutions.

   g. Sufficient feedback for the GA. This is related to the difficulty with Boolean variables and non-arithmetic predicates.

6. We find these factors to be related in some sense. It is reasonable to expect (though not always the case) that a large program would have non-linear

predicates, high structural complexity, long program paths, and complicated data flows. High structural complexity and strict testing criterion increase the number of constraints that a test input must satisfy. Since one might expect to find fewer highly specialized test solutions in an input space, factors c and d contribute to factor e, small disconnect sub-domains. Having some idea about the size of the solution space is only one side of the story. The other is, knowing the size of the search space. This is where factor f, solution density, comes into the picture. Finally, GA search can deteriorate to a random search if it is not receiving the right kind and amount of feedback.

7. The complex nature of programs makes it difficult to identify and quantify exactly what conditions make GA testing outperform random testing consistently. As a result, we have seen that empirical results sometimes contradict each other. This inconsistency could also be due to incomplete analysis, i.e. not considering all factors or focusing on only one factor; and may be even due to the GA approach itself.

Chapter 4 Formal Concept Analysis

Formal Concept Analysis (FCA) is a data analysis technique based on formation of *concepts* on a relation (Ganter 1999). A concept is a maximal grouping of objects with common attributes. The concepts are partially ordered and form a lattice. (FCAHome) reports that FCA has been employed in many fields, such as "medicine and psychology, musicology, linguistic databases, library and information science, software re-engineering, civil engineering and ecology". We use FCA to organize tests according to execution branch commonality, as in (Ball 1999). Tests in the concept showing the most promise in the environment at hand have a chance at being selected as parents. To our knowledge, application of FCA as a GA fitness cum selection function is novel.

4.1 Definition

For a given binary relationship $R \subseteq O$ x A where O is the object set and A is the attribute set, a concept is a pair (O, A) with O $\subseteq O$ and A $\subseteq A$ if and only if A is the maximal set of attributes that apply to all objects in O and O is the maximal set of objects that have all attributes in A. Intuitively, FCA is the theory of maximal rectangles, modulo row and column permutations.

Table 4.1 is a relation with 4 objects and 6 attributes. An 'X' indicates presence of a relation. FCA defines 8 concepts for this relation (Table 4.2) which are arranged in a lattice in Figure 4.1. The pair ({o4}, {a6}) in Table 4.1, is not a concept because {a6} is not the maximal set of attributes that o4 has. The maximal set of attributes for o4 is {a2, a5, a6}. The pair ({o1, o2, o4}, {a5}) is a concept because only objects o1, o2 and o4 have attribute a5 in common, i.e., no other object has attribute a5.

Formally, let σ (O) denote the set of common attributes for any set of objects $O \subseteq O$ such that σ (O) = { a $\in A$ | $\forall$ (o $\in$ O) (o, a) $\in R$ } and let τ (A) define the set of common objects for any set of attributes $A \subseteq A$ such that τ (A) = { o $\in O$ | $\forall$ (a $\in$ A) (o, a) $\in R$ } then a pair (O, A) is a concept if and only if A = σ (O) $\wedge$ O = τ (A). For a concept c = (O, A), extent(c) = O and intent(c) = A.

Table 4.1 A relation table

	Attributes					
Objects	a1	a2	a3	a4	a5	a6
o1		X			X	
o2	X		X		X	
o3	X		X	X		
o4		X			X	X

Table 4.2 Concepts for relation in Table 4.1

Concept	Extent	Intent	Direct Super Concept
C0	-	a1, a2, a3, a4, a5, a6	1, 2, 4
C1	o3	a1, a3, a4	3
C2	o2	a1, a3, a5	3, 6
C3	o2, o3	a1, a3	7
C4	o4	a2, a5,a6	5
C5	o1, o4	a2, a5	6
C6	o1, o2, o4	a5	7
C7	o1, o2, o3, o4	-	-

The set of concepts form a partial order. Concept $c_1 = (O_1, A_1)$ is a sub-concept of $c_2 = (O_2, A_2)$ or c_2 is the super-concept of c_1 if $O_1 \subseteq O_2$ or $A_2 \subseteq A_1$. In Table 4.2, C1 is a sub-concept of C3 because extent (C1) $\subseteq$ extent (C3). Alternatively, C3 is a super-concept of C1 because intent (C3) $\subseteq$ intent (C1). The subsume relationship between concepts forms a concept lattice. The join of two concepts is the intersection of their extents $(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma (O_1 \cap O_2))$ and the meet of two concepts is the intersection of their intents $(O_1, A_1) \vee (O_2, A_2) = (\tau (A_1 \cap A_2), A_1 \cap A_2)$.

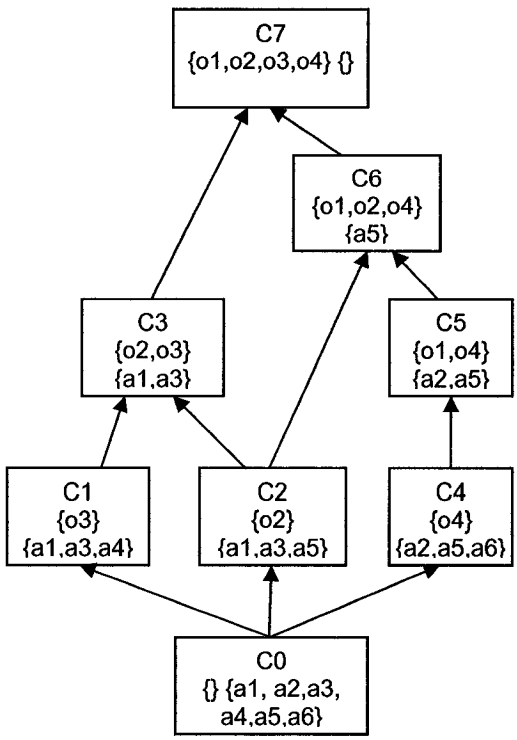


Figure 4.1 The concept lattice for the concepts in Table 4.2

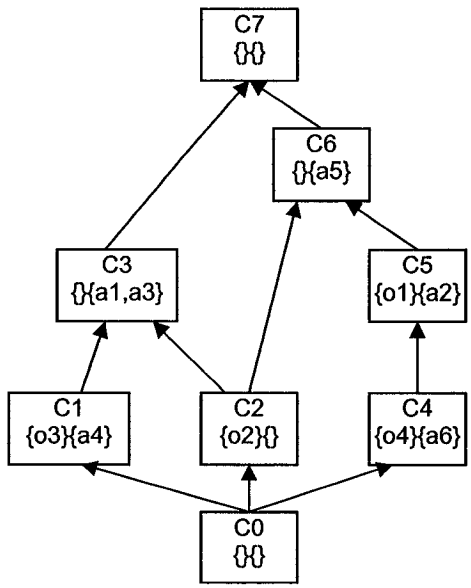


Figure 4.2 The reduced labeled concept lattice of Figure 4.1

To improve readability, attribute labels are pushed up so that the most general concept having attribute a in its intent is marked with a, while object labels are pushed down so that the most specific concept having object o in its extent is marked with o. Thus the (unique) lattice element labeled with a is: $\mu(a) = \vee \{c \in L(C) \mid a \in \text{intent}(c)\}$. Similarly, the (unique) lattice element labeled with o is: $\gamma(o) = \wedge \{c \in L(C) \mid o \in \text{extent}(c)\}$. Figure 4.2 is the reduced labeled version of Figure 4.1. The extent and intent of a concept is reconstructed by having a concept "inherit" all the objects of its direct and indirect sub-concepts and all the attributes of its direct and indirect super-concepts.

There are a number of algorithms and tools available to compute concepts and draw concept lattices (FCAHome). Our implementation (**genet**) uses TKConcept by Lindig (1999).

4.2 genet's fitness cum selection algorithm

In our application of FCA, the object set O is a set of test T and the attribute set A is a set of true-false branches B. A pair (T, B), with $T \subseteq T$ and $B \subseteq B$, is a concept if every test in T exercises every branch in B and branches common to every test in T are all in B, i.e., there are no branches not in B that are exercised by every test in T. The idea is to organize tests according to execution branch commonality. This information is then used to evaluate the fitness of sets of test. The fitness value (rank) is used to select the set of tests with the most potential to produce fit offspring. A "fit" offspring is a test that exercises one or more branches that have not been covered. **genet**'s objective is to exercise branches efficiently. To meet this objective, we select tests from a concept with the highest rank. Alternative decisions include selecting tests from the lowest non-zero ranking concept, and considering the size of a concept's extent together with its rank.

These alternative designs could produce different search qualities. The following example illustrates **genet**'s fitness and selection function.

```
INPUT (X, Y, Z)
BEGIN
 IF (predicate1) {
  PRINT "1T"
  IF (predicate2)
   PRINT "2T"
  ELSE
   PRINT "2F"
 }
 ELSE {
  PRINT "1F"
  IF (predicate3)
   PRINT "3T"
  ELSE {
   PRINT "3F"
   IF (predicate6)
    PRINT "6T"
   ELSE
    PRINT "6F"
   RETURN }
 }
 IF (predicate4)
  PRINT "4T"
  IF (predicate5)
   PRINT "5T"
  ELSE
   PRINT "5F"
 ELSE
  PRINT "4F"
END
```

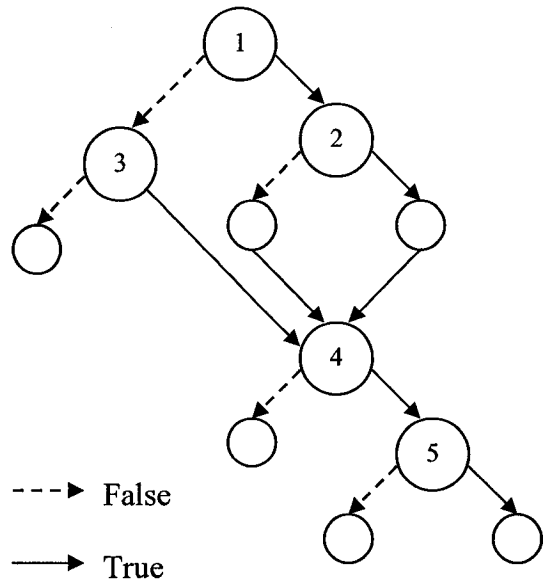


Figure 4.3 A program and its 'dynamic' control flow graph

Table 4.3 Population of tests at generation t

	Branches									
Tests	1T	1F	2T	2F	3T	3F	4T	4F	5T	5F
T1		x			x		x			x
T2	x			x			x			x
T3	x		x				x			x

Table 4.4 Concepts for the relation in Table 4.3

Concept	Extent	r.l. Intent	Direct Super Concept	Rank
C0	-	-	1, 2, 4	0
C1	T3	2T	3	0
C2	T2	2F	3	0
C3	T2, T3	1T	5	0
C4	T1	1F, 3T	5	1
C5	T1, T2, T3	4T, 5F	-	2

At generation t, there are three tests in the population for the program in Figure 4.3 with branch executions as shown in Table 4.3. An 'x' denotes execution. Branches 3F, 4F and 5T have not been exercised yet. C1 in Table 4.4 says that branch 2T is exercised by tests T3 only. C4 says only test T1 have branches 1T and 3T in common. That 4T and 5F are attributes of the top concept lattice element, C5, tells us that all tests exercise branches 4T and 5F. From the situation at generation t, we deduce that branches 4T and 5F have a high affinity with each other; as do 1F and 3T.

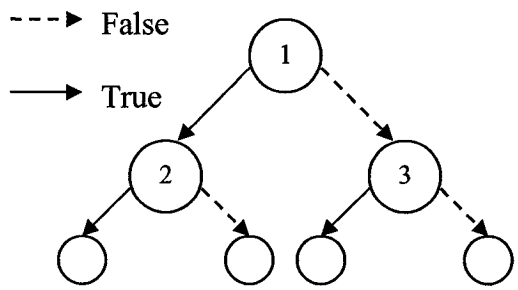
We assume that a test which executes one branch of a predicate is more likely, with adaptation, to execute the other branch of the predicate, than a test which does not visit the predicate at all. For example, it is more likely for an adaptation of T1 to cover 3F than it is for an adaptation of T2 or T3 since neither T2 nor T3 visit or come close to predicate 3 in their execution paths. On the basis of this assumption, C4 is given a rank of 1, denoting that the tests in C4 have potential to cover one uncovered branch (3F). C5 has a rank of 2 because its tests have potential to cover 4F and 5T. In other words, because T1, T2 and T3 already visit predicates 4 and 5 in their execution, we assume that these tests contain suitable base elements for creating tests that will cover 4F and 5T. **genet** selects tests from C5 to be parents of generation t + 1. C5 is the *winning concept* and the actual targets are 4F and 5T.

4.3 Design of the algorithm

We consider several interesting situations for **genet**'s fitness cum selection algorithm. Specifically, we want to understand why FCA is different from ranking each test individually and we want to know if branches with low affinity with each other can appear in a concept. We also describe three unavoidable "problematic" situations.

4.3.1 Opposite edges

Consider the control flow graph and the test executions in Figure 4.4. The uncovered branches, 2T and 3F, lie on extreme opposite edges of the graph. They have very low affinity. There is a tie between the ranks of C1 and C2 (Table 4.5). We break this tie by choosing the concept with the smallest number, i.e. C1. Section 4.3.4 explains this design decision.



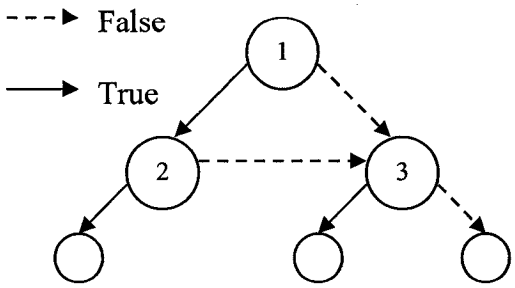
	Branches					
Test	1T	1F	2T	2F	3T	3F
T1	x			x		
T2		x			x	
T3	x			x		

Figure 4.4 Control flow tree and its test executions

Table 4.5 Concepts for relation in Figure 4.4

Concept	Extent	Intent (unique)	Direct Super Concept	Rank
C0	-	-	1, 2	0
C1	T2	1F, 3T	3	1
C2	T1, T3	1T, 2F	3	1
C3	T1, T2, T3	-	-	0

Suppose there is a winning concept (Ci) with attributes 2F and 3T. By definition of a concept, there must be a test (Tj) that exercises both 2F and 3T. But to reach 2F and 3T, Tj must be able to exercise both 1T and 1F. This is a contradiction. By our assumption about the behavior of a program under test (Section 3.3), Tj can only make predicate 1 evaluate to either true or false, not both. A test can exercise both 2F and 3T if Figure 4.5 is the control flow graph. In this case, **genet** chooses C2 as the winning concept (Table 4.6) and targets to cover 2T.



	Branches					
Test	1T	1F	2T	2F	3T	3F
T1	x			x	x	
T2		x			x	
T3		x			x	

Figure 4.5 Control flow tree and its test execution

Table 4.6 Concepts for relation in Figure 4.5

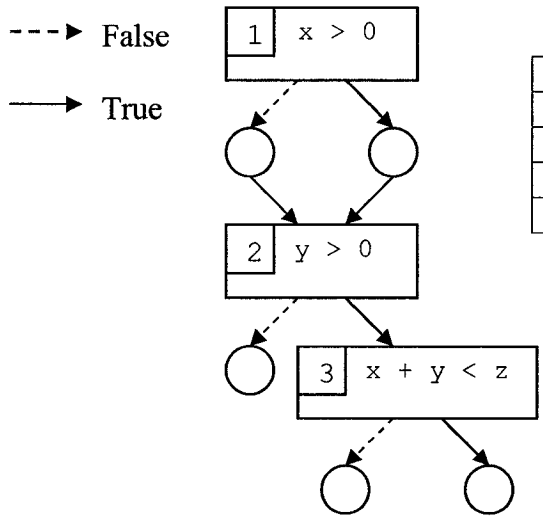
Concept	Extent	Intent (unique)	Direct Super Concept	Rank
C0	-	-	1, 2	0
C1	T2, T3	1F	3	0
C2	T1	1T, 2F	3	1
C3	T1, T2, T3	3T	-	1

4.3.2 FCA and unique (reduced labeled) intents

We compare **genet**'s evaluation of the test population in Figure 4.4 with a non-FCA method. The non-FCA method ranks tests individually, sorts the tests according to their rank values and selects tests with suitably high rank values. With the non-FCA method, T1, T2 and T3 have the same rank value of 1. The non-FCA method suggests that any combination of T1, T2 and T3 is a suitable selection of parents to create fit test offspring. For the situation in Figure 4.4, it is clear that a fit test offspring can cover only one of the two target branches at a time, not both. So it is not sensible to mix the genes of T1 and T2 or of T2 and T3 since T1 and T3 traverse an identical execution path which is the exact opposite of the path for T2. We know about this similarities and differences of execution paths of the tests from the information generated by FCA (Table 4.5).

Without unique intents, the highest ranking concepts (excluding the bottom element whose attributes consists of all covered branches) will include those describing complete execution paths so that selection of parents is likely to be confined to tests that execute

the same path. However this may not always be the best course of action, as illustrated by the example in Figure 4.6.



	Branches					
Test	1T	1F	2T	2F	3T	3F
T1		x		x		
T2	x		x			x
T3		x	x			x

Figure 4.6 A control flow tree and its test executions

Table 4.7 Concepts for relation in Figure 4.6

Concept	Extent	Intent (non-unique)	Rank1	Intent (unique)	Rank2	Direct Super Concept
C0	-	1F, 1T, 2F, 2T, 3F	-	-	-	1, 2, 4
C1	T3	1F, 2T, 3F	1	-	0	3, 5
C2	T2	1T, 2T, 3F	1	1T	0	3
C3	T2, T3	2T, 3F	1	2T, 3F	1	6
C4	T1	1F, 2F	0	2F	0	5
C5	T1, T3	1F	0	1F	0	6
C6	T1, T2, T3	-	0	-	0	-

In Table 4.7, the bottom element is C0 and concepts whose (non-unique) intent describes complete execution paths are C1, C2 and C4. Rank1 values are evaluated based on non-unique intents and Rank2 values are evaluated based on unique intents. When Rank1 values are used, C1, C2 and C3 are the highest ranked concepts. When Rank2 values are used, C3 is the highest ranked concept.

Selecting T2 and T3 as parents to create a test for 3T is a more suitable choice than selecting either T2 or T3 alone because mixing the genes of T2 and T3 give more

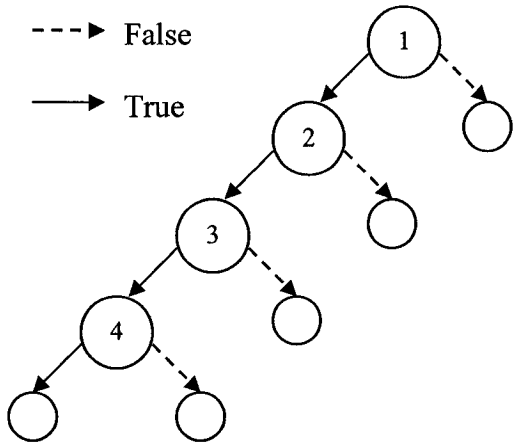
combinations that satisfy condition 3, $(x + y < z)$, than the genes of either T2 or T3 alone. In T2, both the x and y genes are positive. In T3, the x gene is negative and the y gene is positive. So for a domain of -5 to 5 for each of the input variables, having both T2 and T3 in the parent pool increases the chance of obtaining a combination of x, y and z genes that satisfy the constraint for 3T, i.e. $(x + y) < z$.

Note that the presence of concepts that do not describe complete execution paths, i.e. C3 and C5, are only possible if there are at least two tests in the population with similar but non-identical execution paths.

4.3.3 Non unique extents.

The most promising or winning concepts in the examples so far have either been a top lattice element or a direct sub-concept of the top element. The extent of a top element always includes all objects in the relation, so we ignored the second part of the selection algorithm in previous examples.

The second part of the selection algorithm is gathering and scoring tests from the ancestors of the winning concept. The ancestors of a concept are all its direct and indirect super concepts. The reason for including this additional step is to handle the situation when the size of the winning concept's extent is smaller than the size of the parent population. Tests from ancestor concepts will have more similarities with tests in the winning concept than test selected or generated at random. Scoring tests is counting the frequency that each test appears along the lattice path from the winning concept to the top element. To enable scoring, the extent obviously cannot be unique. Scoring increases the selection likelihood of tests from the winning concept and from concepts closer to the winning concept than from concepts farther away from the winning concept.



	Branches							
Test	1T	1F	2T	2F	3T	3F	4T	4F
T1		x						
T2	x		x		x			x
T3	x		x			x		

Figure 4.7 A control flow tree and its test execution

Table 4.8 Concepts for relation in Figure 4.7

Concept	Extent	Intent (non-unique)	Intent (unique)	Rank	Direct Super Concept
C0	-	1F, 1T, 2T, 3F, 3T, 4F	-	-	1, 2, 4
C1	T3	1T, 2T, 3F	3F	0	3
C2	T2	1T, 2T, 3T, 4F	3T, 4F	1	3
C3	T2, T3	1T, 2T	1T, 2T	1	5
C4	T1	1F	1F	0	5
C5	T1, T2, T3	-	-	0	-

In the example of Figure 4.7 and Table 4.8, the winning concept is C2 and its ancestors are C3 and C5. Scoring tests gives 1, 3 and 2 for T1, T2 and T3 respectively. T2 gets the highest score because it appears in the extents of the winning concept and its direct super concept. The score for T3 is higher than T1 because T3 appears in the winning concept's direct super concept and T1 does not. So the scores for tests in the winning concept or in concepts closer to the winning concept are higher. These test scores are sorted and selection preference is given to tests with the highest scores.

4.3.4 Smaller concept number.

When two or more concepts have the same highest rank, we break the tie by choosing the concept with the smallest number. One alternative is to choose the largest number. However concepts nearer to the bottom of a concept lattice tend to have larger (non-unique) intents. The concept at the bottom of the lattice has the largest intent. Therefore, to maximize branch coverage, it makes sense to break a tie by choosing the concept with the smallest number. In Table 4.8, the two concepts with the highest score are C2 and C3, and C2 is a sub concept of C3. **genet** chooses C2 as the winning concept.

4.3.5 Unknown predicates.

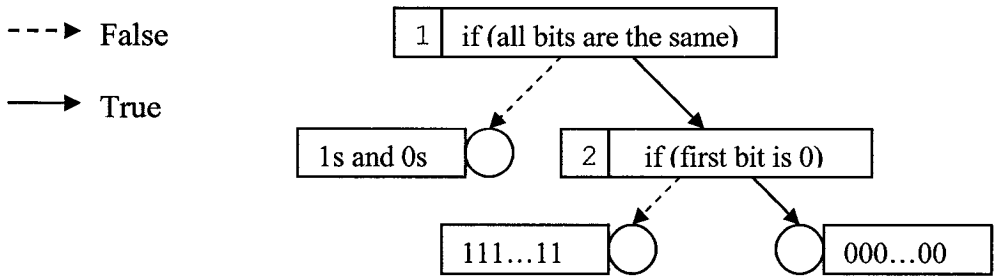
In all the examples so far, either one or both of the branches of a predicate is covered so that **genet** is more or less aware of how all predicates hang together. However this may not always be the case. **genet** is not fed complete knowledge of the structure of the program so it does not know how the predicates relate to each other until it has found a test that makes a connection between predicates.

In Figure 4.3, 3F actually leads to predicate 6 so that in trying to cover 3F, **genet** would actually be trying to cover three branches, 3F, 6T and 6F. Instead, because **genet** does not yet know where 6T and 6F fit into the structure of the program, it decides to target 4F and 5T by selecting C5 as the winning concept.

4.3.6 Deception.

genet makes selection based on dynamic control flows. It has no knowledge of the semantics of the predicates. Following is a situation where **genet** can be "deceived".

Suppose all branches for the control flow graph in Figure 4.8 are covered except for 2T. Then **genet** will select C1 as the winning concept. But tests from C1 are strings of all 1's while **genet** is trying to compose a string of all 0's to cover 2T.



	Branches			
Test	1T	1F	2T	2F
T1		x		
T2	x			x

Concept	Extent	Intent (unique)	Direct Super Concept	Rank
0	-	-	1, 2	-
1 (C1)	T2	1T, 2F	3	1
2	T1	1F	3	0
3	T1, T2	-	-	-

Figure 4.8 A deceptive control flow tree, its test population and concept table

4.3.7 Unreachable branches.

genet assumes that all program branches and predicates are reachable. When this is not the case, the unreachable branch or predicate could misdirect **genet**'s efforts.

Chapter 5 A closer look at genet

This chapter gives an overview of **genet**'s design and implementation, and shows how **genet** is to be used with an actual program.

5.1 Algorithm

Inputs (see Appendix A for examples):

1. an executable program under test, instrumented to record triggered branches,
2. a *TestGenome* class which defines the test chromosome, command to run the program with a test, crossover operator and mutation operator, and
3. an *EvolTestDriver* class which specifies the number of predicates to cover (*predSz*), size of the initial population (*initpopSz*), size of the parent pool (*parentSz*), crossover rate (*Xr*), mutation rate (*Mr*), life span of a chromosome (*Clf*), maximum number of generations (*Gmax*) and number of generations **genet** tolerates an unchanged target before introducing random chromosomes into the next generation (*Tmax*). *Tmax* = 0 turns this feature off, i.e. **genet** will not introduce random chromosomes into generations post initial population.
4. Create initial tests. This is optional. **genet** accepts an empty test input file and produces enough random test chromosomes to form an initial population of *initpopSz* size.

Outputs:

1. test data,
2. branch coverage and
3. concepts relating tests and branches

The flow chart in Figure 5.1 tells the story of how **genet** works. Figures 5.2 and 5.3 display the corresponding code snippets. Appendix B treats the implementation of **genet** (and **randy**, our random test generator) more fully.

When **genet** runs a test program with a test, trace data is produced to indicate which branches in the test program have been triggered by the test. **genet** uses this trace data

information to update its decision table. After running every test in a generation, **genet** does a calculation on its decision table to find out what if any, are the remaining uncovered branches. If there are still a few uncovered branches, **genet** submits all tests with a positive lifespan to formal concept analysis (FCA). **genet**'s genetic algorithm (GA) component uses the information produced by FCA to create the next generation. Every time a test is selected by **genet** to be a parent, its lifespan deceases by one. Essentially, **genet**'s GA takes care of the fitness and selection functions (which were described in Chapter 4) but relies on the crossover and mutation operators defined in the *TestGenome* class. This process is repeated until either all branches are covered or the maximum number of generations is reached. **genet** may terminate prematurely if it cannot find any suitable parents.

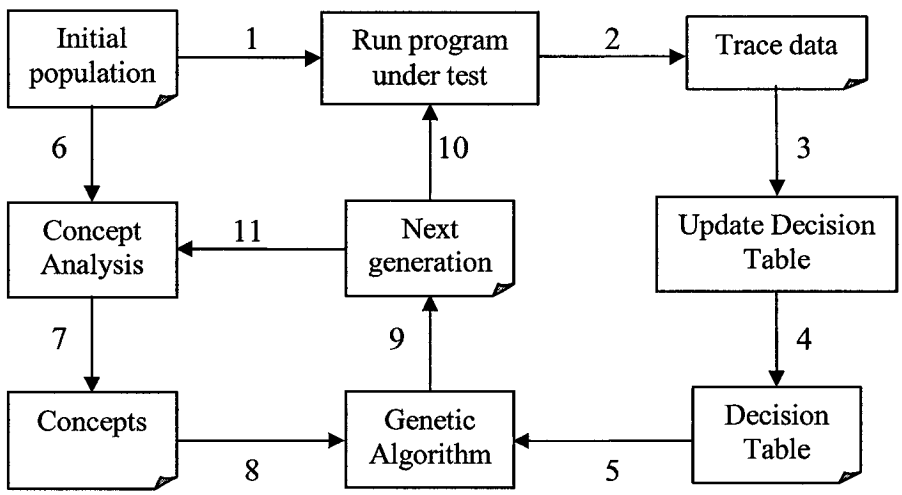


Figure 5.1 Flow chart describing genet.

```
// public
template <class TestGenome, class FitnessEvaluator>
void EvolDATG<TestGenome, FitnessEvaluator>::run() {
  initialize_population();
  for (int i = 0; i < new_tests.size(); i++) {
    if (new_tests[i].evaluate(fitness_evaluator))
      min_tests.push_back(new_tests[i]); // test covers a new branch
    all_tests.push_back(new_tests[i]);
  }
  int times = 0; // number of times score did not change
  int score = fitness_evaluator.score();
  fitness_evaluator.print();
  while ((current_iteration_d < max_iterations_d) &&
      (! fitness_evaluator.done())) {
    fitness_evaluator.evaluate(all_tests);
    evolve_next_generation();
    for (int i = 0; i < new_tests.size(); i++) {
      if (new_tests[i].evaluate(fitness_evaluator))
        min_tests.push_back(new_tests[i]);
      all_tests.push_back(new_tests[i]);
    }
    if (fitness_evaluator.score() == score) {
      times++;
      if (times > max_tries) {
           times = 0; introduce_new_individuals = true;
      }
    }
    else {
      times = 0; introduce_new_individuals = false;
    }
    score = fitness_evaluator.score();
    fitness_evaluator.print();
    ++current_iteration_d;
  }
}
```

```
// private
template <class TestGenome, class FitnessEvaluator>
void EvolDATG<TestGenome, FitnessEvaluator>::evolve_next_generation() {
  new_tests.clear();
  parents.clear();
  fitness_evaluator.getNfittest(ntests_per_iteration_d, parents);
  if (parents.size() > 0) {
    for (int i = 0; i < parents.size(); i++)
      all_tests[parents[i]].age();
    recombine();
    mutate();
  }
  if (introduce_new_individuals)
    introduce();
}
```

Figure 5.2 Code snippets from EvolDATG class

```
// public
// evaluate tests that can reproduce, i.e. life > 0
template <int NPREDICATES, class TestGenome>
void FCA_FitnessEvaluator<NPREDICATES,
TestGenome>::evaluate(vector<TestGenome>& all_tests) {
  relation.clear();
  for (int i = 0; i < all_tests.size(); i++)
    if (all_tests[i].life() > 0)
      relation.insert(pair<int, const char*>(i,
                                        all_tests[i].behavior()));
   if (relation.size() > 0) {
    concept_analyzer.analyze(relation);
    load_concepts();
    rank_concepts();
    score_tests();
  }
}
```

```
// public
template <int NPREDICATES, class TestGenome>
void FCA_FitnessEvaluator<NPREDICATES, TestGenome>::getNfittest(int n,
vector<int>& selection) {
  int k = 0;
  multimap<int, int>::reverse_iterator hist_rit = histogram.rbegin();
  while ((hist_rit != histogram.rend()) && (k < n)) {
    selection.push_back(hist_rit->second);
    hist_rit++; k++;
  }
}
```

```
// private
template <int NPREDICATES, class TestGenome>
void FCA_FitnessEvaluator<NPREDICATES, TestGenome>::rank_concepts() {
  bitset<NPREDICATES> target_predicates = branch_coverage.target();
  for (int i = 0; i < concepts.size(); i++) {
    bitset<NPREDICATES> res = concepts[i].attributes_bitset();
    res &= target_predicates;
    int r = res.count();
    concepts[i].rank(r);
    if ((r > highest_rank_d) || (highest_rank_d == -1)) {
      winning_concept_d = concepts[i].conceptid();
      highest_rank_d = r;
    }
  }
}
```

Figure 5.3 Code snippets from FCA_Fitness Evaluator class

```
// private
template <int NPREDICATES, class TestGenome>
void FCA_FitnessEvaluator<NPREDICATES, TestGenome>::score_tests() {
  map<int, int> m;
  // <test_id(asc), 1>

  vector<int> v1 = concepts[winning_concept_d].objects();
  for (int i = 0; i < v1.size(); i++)
    m[v1[i]] += 1;

  // <test_id(asc), freq>
  vector<int> v2 = concepts[winning_concept_d].uppers();
  for (int j = 0; j < v2.size(); j++) {
    vector<int> v3 = concepts[v2[j]].objects();
    for (int k = 0; k < v3.size(); k++) {
      m[v3[k]] += 1;
    }
  }
  // fill histogram with <freq(asc), test_id>
  histogram.clear();
  map<int, int>::iterator mit = m.begin();
  while (mit != m.end()) {
  // cout << "   " << mit->first << "  " << mit->second << endl;
    histogram.insert(pair<int,int>(mit->second , mit->first));
    mit++;
  }
}
```

Figure 5.3 (cont'd) Code snippets from FCA_Fitness Evaluator class

5.2 An example

The Remainder program (Sthamer 1995) (see Appendix B for the code) accepts two integer inputs, divides the first by the second and outputs the remainder, if any. A remainder value is -1 when a divisor is 0 or a dividend is smaller than its divisor. Figure 5.4 is Remainder's loop-free control flow graph. A node's label identifies the predicate it represents. A true branch from a node with label *i* is denoted *iT*. Loops may be handled in **genet** by unrolling them as is done in (Jones 1996). The initial population consists of three tests (Table 5.1). The test interval is [-10, 10]. Table 5.2 is the decision table after evaluating the initial population of tests for Remainder. Table 5.3 is the concepts and their rank for the initial population. $Gmax = 10$ and $Clf = 1$ for Remainder.

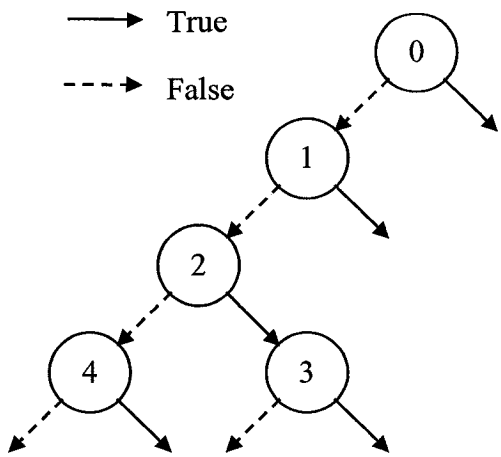


Figure 5.4 Control flow tree for Remainder program

Table 5.1 Initial population of tests

Test	Dividend	Divisor	Result	Remainder	Trace	Clf
T0	9	-2	-4	1	0F 1F 2T 3F	1
T1	1	3	0	-1	0F 1F 2T 3T	1
T2	-5	0	0	-1	0F 1T	1
T3	4	-7	0	-1	0F 1F 2T 3F	1
T4	8	6	1	2	0F 1F 2T 3T	1
T5	6	4	1	2	0F 1F 2T 3T	1

Table 5.2 Decision table after evaluating the initial population

Predicate	True	False	Target
0	1	0	1
1	0	0	0
2	0	1	1
3	0	0	0
4	1	1	1

The number of rows in **genet**'s decision table is determined by *predSz*. '0' indicates coverage. A predicate is *partially covered* if either its true or false branch is covered. Table 5.2 says that branches 0T, 2F, 4T and 4F are uncovered while the rest of the branches are covered. It also tells us that predicates 0 and 2 are partially covered, predicates 1 and 3 are fully covered and predicate 4 is completely uncovered. **genet** uses three bit vectors to implement its decision table. The rightmost bit of a vector corresponds to predicate 0 and a clear bit in a decision table bit vector denotes coverage

of the branch or predicate. So the true and false columns of Table 5.2 are represented in **genet** as 10001 and 10100. The target bit vector is calculated by OR-ing the true and false bit vectors. It serves as a mask to rank the concepts.

Table 5.3 Concepts for the initial population

Concept	Extent	r.l. Intent	Intent as bit vector	Direct Super Concepts	Rank
C0	-	-	00000	C1, C2, C3	0
C1	T2	1T	00010	C5	0
C2	T0	3F	01000	C4	0
C3	T1, T3, T4, T5	3T	01000	C4	0
C4	T0, T1, T3, T4, T5	1F, 2T	00110	C5	1
C5	T0, T1, T2, T3, T4, T5	0F	00001	-	1

The rational behind concept ranking was explained in Chapter 4 and is implemented as the *FCA_FitnessEvaluator::rank_concepts* method (Figure 5.3) in **genet**. To enable concept rank calculation, the reduced labeled (r.l.) concept intents are converted into attribute bit vectors (Table 5.3). A clear bit in an attribute bit vector means the branch represented by the bit is not covered by the set of tests in the corresponding extent. Concept rank is calculated by taking the intersection of (AND-ing) the attribute bit vectors with the current target bit vector. The current target bit vector is 10101. Therefore concepts C4 and C5 get a rank of 1 while the other concepts get a rank of 0. When there is more than one concept with the highest rank, the concept with the lower number wins (per explanation in Chapter 4). This makes C4 the winning concept and parent selection will begin with tests in this concept. For programs with multiple exits, it is possible for no concept to have a positive rank, in which case either the predicate(s) in question is actually unreachable or **genet** is unable to find suitable test data for it (them).

Having found the highest ranked concept, tests are scored (as explained in Chapter 4). This entails bagging the tests from C4 and its super concepts (C5), and then tabulating

the frequency of each test that appears in this bag. The *FCA_FitnessEvaluator::score_tests* method (Figure 5.3) accomplishes the test scoring task in **genet**. The bag of tests for Remainder's next generation contains T0, T1, T3, T4 and T5 (from C4) and T0, T1, T2, T3, T4 and T5 (from C5). So the score for T0, T1, T3 and T4 is 2 each and T2 gets a score of 1. Scoring imposes an order on the tests selected. The tests are ordered by their frequency and this histogram is used to select the parents for the next generation of tests. The parent pool is formed by selecting *parentSz* tests starting from the fittest end of the histogram. For Remainder, *parentSz* = 3. So the parent pool for the next generation consists of T0, T1 and T3. The lifespan of T0, T1 and T3 is reduced by 1.

The next generation of tests is created by randomly selecting pairs of tests from the parent pool to recombine and randomly selecting individual tests from the parent pool to mutate. Recombination takes place ($Xr \times parentSz$) times and mutation occurs ($Mr \times parentSz$) times. Recombination only occurs between parents with non-identical genes. A recombination produces two offspring, one of which is discarded if the two offspring have identical genes. So it is possible for the size of the next generation to be slightly smaller than ($Xr \times parentSz \times 2 + Mr \times parentSz$). For the Remainder program, Xr = 0.8 and Mr = 0.4. Thus, evolution is by doing crossover twice and mutation once; and the number of tests produced for the next generation is at most five $\lfloor 0.8 \times 3 \times 2 + 0.4 \times 3 \rfloor = 5$.

RemainderTestGenome uses real value representation, "shuffle" crossover and uniform mutation. A test is a two-integer chromosome. The index of the left gene is 0. **genet** randomly pairs up T0 with T3 and T1 with T3 for recombination. The "shuffle"

crossover exchanges a randomly selected gene from a chromosome for another randomly selected gene from its partner chromosome. The crossover between T0 (9, -2) and T3 (4, -7) produces T6 (-7, -2) and T7 (4, 9). The crossover between T1 (1, 3) and T3 (4, -7) produces T8 (1, 4) and T9 (3, -7). **genet** randomly selects T0 for mutation and T0 (9, -2) mutates to T10 (3, -2). T6, T7, T8, T9 and T10 makes generation 1. The initial population is generation 0.

genet runs Remainder with each of the test in generation 1 and if the decision table is not completely covered at the end of the run, individuals of generation 1 are inserted into the population. Table 5.4 is the decision table after evaluating generation 1. Table 5.5 shows the population at generation 1. **genet** excludes T0, T1 and T3 from participating in concept analysis and subsequent selection process because their *Clf* values are less than 1. Table 5.6 shows the concepts of population at generation 1.

Table 5.4 Decision table after evaluating generation 1

Predicate	True	False	Target
0	1	0	1
1	0	0	0
2	0	0	0
3	0	0	0
4	1	0	1

Table 5.5 Population of tests at generation 1

Test	Dividend	Divisor	Result	Remainder	Trace	Clf
T0	9	-2	-4	1	0F 1F 2T 3F	0
T1	1	3	0	-1	0F 1F 2T 3T	0
T2	-5	0	0	-1	0F 1T	1
T3	4	-7	0	-1	0F 1F 2T 3F	0
T4	8	6	1	2	0F 1F 2T 3T	1
T5	6	4	1	2	0F 1F 2T 3T	1
T6	-7	-2	3	1	0F 1F 2F 4F	1
T7	4	9	0	-1	0F 1F 2T 3T	1
T8	1	4	0	-1	0F 1F 2T 3T	1
T9	3	-7	0	-1	0F 1F 2T 3F	1
T10	3	-2	-1	1	0F 1F 2T 3F	1

Table 5.6 Concepts of population at generation 1

Concept	Extent	r.l. Intent	Intent as bit vector	Direct Super Concepts	Rank
C0	-	-	00000	C1, C2, C3, C4	0
C1	T2	1T	00010	C7	0
C2	T6	2F, 4F	10100	C6	1
C3	T9, T10	3F	01000	C5	0
C4	T4, T5, T7, T8	3T	01000	C5	0
C5	T4, T5, T7, T8, T9, T10	2T	00100	C6	0
C6	T4, T5, T6, T7, T8, T9, T10	1F	00010	C7	0
C7	T2,T4, T5, T6, T7, T8, T9, T10	0F	00001	-	1

C2 is the winning concept. Scores for tests are T6 = 3, T4 = 2, T5 = 2, T7 = 2, T8 = 2, T9 = 2 and T10 = 1. Tests T6, T4, T5 are selected as parents. The evolution process described previously creates generation 2: T11 (-7, 8), T12 (-2, 6), T13 (6, 6), T14 (8, 4) and T15 (0, 4). After running generation 2, **genet** finds its decision table is completely covered and terminates its execution. Table 5.7 is the final population for Remainder.

Table 5.7 Population of tests at generation 2

Test	Dividend	Divisor	Result	Remainder	Trace	Clf
T0	9	-2	-4	1	0F 1F 2T 3F	0
T1	1	3	0	-1	0F 1F 2T 3T	0
T2	-5	0	0	-1	0F 1T	1
T3	4	-7	0	-1	0F 1F 2T 3F	0
T4	8	6	1	2	0F 1F 2T 3T	0
T5	6	4	1	2	0F 1F 2T 3T	0
T6	-7	-2	3	1	0F 1F 2F 4F	0
T7	4	9	0	-1	0F 1F 2T 3T	1
T8	1	4	0	-1	0F 1F 2T 3T	1
T9	3	-7	0	-1	0F 1F 2T 3F	1
T10	3	-2	-1	1	0F 1F 2T 3F	1
T11	-7	8	0	-1	0F 1F 2F 4T	1
T12	-2	6	0	-1	0F 1F 2F 4T	1
T13	6	6	1	0	0F 1F 2T 3T	1
T14	8	4	2	0	0F 1F 2T 3T	1
T15	0	4	0	-1	0T	1

Chapter 6 Experiments

The experiments are intended to explore **genet**'s suitability to types of programs by comparing its performances with randy's. A Mersenne random number generator (Agner 2004) is used which produces a uniform distribution and the same sequence of numbers each time. Experiments are repeated five times to obtain a more robust result. Solution density of a search space is defined as number of feasible combinations divided by number of possible combinations. What constitutes a feasible combination depends on the problem at hand.

6.1 Triangle

Triangle is ubiquitous in testing literature. It is the problem of determining if a combination of three integers, each representing a length measurement, makes a triangle, and if it does; what type of triangle. We use the implementation of Triangle found in (McGraw 1998) which has 10 predicates (or 20 branches), and four possible outcomes: not a triangle, scalene, isosceles and equilateral.

Real value representation is used: a Triangle chromosome is an array of three integers. The test interval for each gene is all integers within [-1, 998]. Non-positive integer values are necessary to trigger error situations, e.g. length cannot be zero or negative. Total number of combinations in this test interval is 1000^3 which is a fairly large search space. A feasible combination is a combination that makes a triangle. The solution density is moderate (around 50%). This test interval creates a more favorable environment for randy than say a test interval of [-499, 500] and gives **genet** quite a challenge (since **genet** needs to outperform **randy** to justify its additional complexity).

The parameter values used in this experiment are: initpopSz = 100, parentSz = 10, Gmax = 50, Xr = 0.6, Mr = 0.8, Clf = 2 and Tmax = 0. Maximum number of tests is 1100. Four experiments were made on Triangle:

a. **randy**,
b. **genet** with uniform crossover and uniform mutation,
c. **genet** with shuffle crossover and uniform mutation, and
d. random parent selection (without **genet**) with shuffle crossover and uniform mutation.

The rationale behind these sub-experiments follows. Since **genet**'s fitness function is non-arithmetic, Triangle becomes a combinatorial problem. That is, instead of calculating the difference between a length and the desired length needed to make some triangle combination, **genet** groups chromosomes that satisfy similar constraints (through FCA) and expects similarly grouped chromosomes to produce other kinds of useful chromosomes (test solutions) through the adaptation process. For a constraint like `(a + b > c)` in Triangle to be true, an arithmetic fitness function is able to rate chromosome ⟨4, 4, 9⟩ to be more fit than ⟨4, 4, 11⟩; but **genet** cannot make such distinction and groups the two chromosomes in a concept. This means isosceles chromosomes that traverse the same path through Triangle, such as ⟨4, 4, 5⟩, ⟨4, 4, 7⟩ and ⟨3, 3, 5⟩, will be in the same concept and are likely to appear in the same parent pool. The task of the *TriangleTestGenome* designer then is to construct genetic operators which can transform such a pool of isosceles combinations into other test solutions.

The No Free Lunch (NFL) theorem (Wolpert 1995) makes it unreasonable to expect *any* genetic operator to produce useful adaptations and **genet** to outperform randy at the

same time. For a search (or optimization) algorithm *a* to be effective for a problem (or cost function) *f*; the biases in *a* must match with the particulars of *f*. If no knowledge of the problem is incorporated into a search algorithm, then the NFL theorem demonstrates that there is no reason to believe the search algorithm will be effective. A consequence of the NFL theorem is that it is impossible to say that one algorithm performs better than another algorithm, on average over all cost functions (Wolpert 1995). Thus, for **genet** to operate as an effective general purpose automatic test generator, it is necessary for its genetic operators to be determined by its *TestGenome* designer base on domain knowledge of the program under test.

To construct genetic operators for Triangle, we consider the least probable combination to find by chance, an equilateral combination which we expect[5] will evolve from an isosceles combination. The *TriangleTestGenome* genetic operators should be able to adapt an isosceles into an equilateral. In the example of the three chromosomes (in the previous paragraph), if ⟨4, 4, 5⟩ and ⟨4, 4, 7⟩ are partnered for a crossover; then it is a matter of placing a 4 in the rightmost position of one of the chromosomes without disrupting its other genes. Uniform crossover and mutation is unlikely to achieve this adaptation. "Shuffle" crossover (also used by *RemainderTestGenome* in Chapter 5) is a more suitable operator. A successful "shuffle" might exchange the leftmost gene of the first chromosome with the rightmost gene of the second chromosome, producing ⟨7, 4, 5⟩ and an equilateral combination ⟨4, 4, 4⟩. "Shuffle" is only meaningful if all inputs share the same domain.

[5] Since of all other combinations, an isosceles most closely resembles an equilateral combination. The role of resemblance is discussed later on in this chapter.

To summarize, the objective of sub-experiments b and c are to see the effect of different genetic operators on genet's performance. Sub-experiment d is a control experiment to see whether **genet** (selection pressure) can make a difference to the search performance when suitable genetic operators are used. Table 6.1 reports on the result of this experiment.

Table 6.1 Results for Triangle

	Number of branches & tests				
Seed	a	b	c		d
11	15	8	20	298	18
13	13	17	20	352	19
17	17	8	20	280	19
19	16	11	20	676	15
23	17	12	20	280	16
Average	15.6	11.2	20	377.2	17.4

In the five runs that we did, sub-experiment c had the best performance. Sub-experiment c was the only one to cover Triangle completely and did so quite efficiently, using 378 tests on average. Sub-experiments a, b and d ran to *Gmax* without reaching full branch coverage. Sub-experiment b did worse than randy. Its poor performance is attributed to inappropriate genetic operators. This is supported by sub-experiment c. When uniform crossover was substituted with shuffle, genet's performance improved remarkably. Sub-experiment d did better than randy and sub-experiment b, but not as well as sub-experiment c. This result suggests to us that adaptation plays a more significant role for the Triangle problem than selection; but selection is still useful as an accelerant to the evolution process.

Figure 6.1 summarizes the search progress of the sub-experiments. Altenberg (1995) offers the opinion that one way to evaluate the quality of search of a GA "...is to compare the ability of a GA to generate new, highly fit individuals with the rate at which

they are generated by random search." He introduced the notion of *evolvability* which means "the ability to produce individuals fitter than any existing", as a measure of GA performance. By this performance measure, sub-experiment c has the best evolvability since the rate at which it finds fitter individuals (tests which increase total branch coverage) is greater than random search and than other sub-experiments. The graph for sub-experiment c has the steepest ascent.

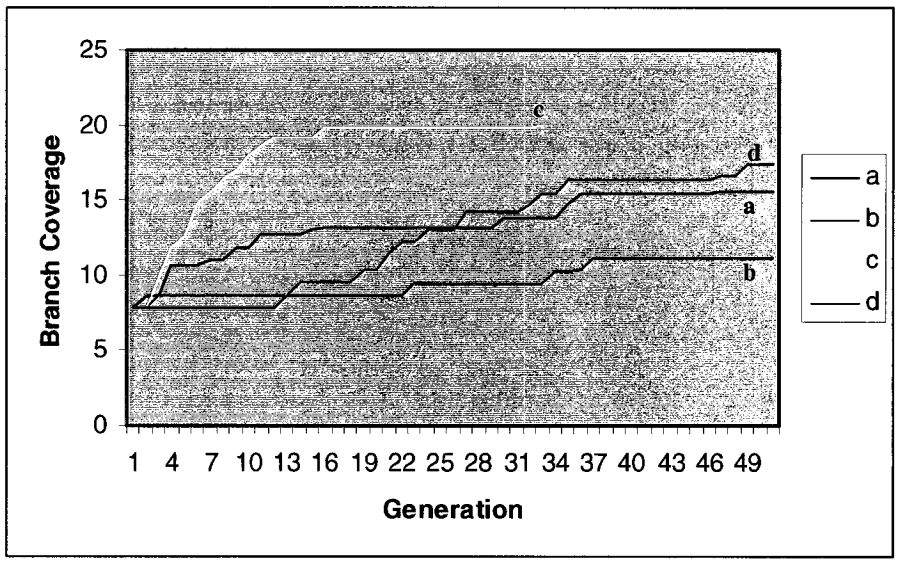


Figure 6.1 Evolutions of Triangle sub-experiments

Sub-experiment b has the least evolvability although it uses **genet** to select useful *building blocks* or *schemas*. The traditional theory of GAs revolves around the Schema Theorem (Holland 1975) and the Building Block Hypothesis (Goldberg 1989). It supposes that the power of GAs lie in their ability to discover, emphasize and recombine good building blocks of solutions (Mitchell 1996, page 27). However, there are opposing views which argue that the appearance of more fit schema is not indicative of the search quality of a GA (Altenberg 1995). Instead, Altenberg (1995) illustrates that evolvability depends on the *transmission function* having knowledge of what the GA is trying to

achieve. Transmission function T (x ← y, z) is defined as "the probability that offspring genotype x is produced by parental genotypes y and z as a result of the action of genetic operators on the representation" (Altenberg 1995).

The results of sub-experiments b and c agree with the theories of Altenberg (1995) and Wolpert (1995). **genet** has the best search performance when its genetic operators are biased to find an equilateral combination (sub-experiment c). When **genet** does not use any domain knowledge (or when *f* does not match *a*) as in sub-experiment b, its performance is even inferior to a random search.

The graph of sub-experiment d begins to ascend rapidly after an initial plateau. The reason for this is the shuffle crossovers in the first 13 generations have helped the population to converge somewhat. Hence after generation 13, it is more likely to find pairs of chromosomes with similar gene values and to produce combinations with two or more genes with the same value at random.

6.2 Tax

Tax is a program written from a flowchart in (Montalbano 1974, page 48). It makes tax filling decisions with four possible outcomes: "Obtain publication 519", "File a tax return", "File for a tax refund" or "Do not file". Tax has 17 predicates and 12 input parameters, 8 of which are Boolean types and the rest are integer types.

The test intervals for the integer type input parameters *Age*, *SpouseAge* and *SelfEmployedIncome* are so specified that they have two sub-domains each. For example, the test interval for *Age* is [45, 85] and all conditions that use *Age* checks to see if *Age* ≤ 65; so *Age*'s test interval is effectively divided into two equally sized portions. *GrossIncome* (*GI*) is a more complicated integer type input parameter. Its test interval is

[0, 4100], roughly divided into six portions. *GI* is involved in two decisions: $GI \leq 600$ and $GI >$ *minTaxableGI*. *minTaxableGI* depends on a taxpayer's marital status, age and spouse's age, and takes one of the following values: 1700, 2300, 2900 and 3500. The total number of effective combinations for Tax is 12,288 (2^{11} x 6). The solution density of this search space is high, all combinations are feasible.

Real representation is used: a Tax chromosome is an array of 12 integers. The parameter values used in this experiment are: initpopSz = 50, parentSz = 10, Gmax = 50, Xr = 0.6, Mr = 0.8, Clf = 2 and Tmax = 0. Maximum number of tests is 1050. Three experiments were made on Tax:

e. **randy**,

f. **genet** with uniform crossover and uniform mutation, and

g. random parent selection (without **genet**) with uniform crossover and uniform mutation.

Tax is a different type of problem from Triangle. Its search space is much smaller and more feasible than Triangle's. Tax makes many Boolean valued comparisons, which would be difficult to handle with an arithmetic fitness function. Moreover, its inputs do not share the same domain, its most difficult combination is not readily identifiable since complexity of a predicate is not always obvious, its genes have very low number of alleles – effectively only two each for all but *GI*, and its chromosome is longer – 12 loci.

Test solution resemblance is one important way in which Tax differs from Triangle. Test solutions for Tax are *necessarily* more similar than those for Triangle. In the case of Triangle, low diversity in the parent gene pool helps **genet** to converge as illustrated in sub-experiments c and d; but resemblance between parent and offspring is not necessary.

Any ⟨a, b, c⟩ combination where a = b = c makes an equilateral, so it does not matter, in the combinations that have two same value genes, what values these genes take. This flexibility is not the case in Tax.

A test that wants to reach any of Tax's deeply nested predicates (14, 15 and 16) must first traverse the sub-path ⟨T4, T5, F8, T9, F12, F13⟩. This means the following parameters must have their values as: d = 1, e > 600, f = 1, g = 0 and h = 0. The parameter values of d, e, f, g and h form a building block which is necessary for the formation of tests to cover predicates 14, 15 and 16. That there is a necessary parent-offspring resemblance in Tax suggests the application of traditional GA theory and selection to be more dominant than adaptation.

It is preferable that genetic operators refrain from breaking up fit building blocks so that fit schemas may be reused in subsequent generations. However it is not always easy to identify fit schemas and the fitness of a schema depends on the current target. *TaxTestGenome* uses uniform crossover and uniform mutation. Since tests in a parent pool have some branches in common, a vertical exchange of genes at alternate positions allows **genet** to explore the search space without causing too much disruption to fit blocks. Mutation is applied uniformly across the Tax chromosome because the number of alleles for each parameter is near uniform.

The results from five runs reported in Table 6.2 supports our hypothesis. Sub-experiment f outperforms sub-experiments e and g quite significantly even when run with seed number 17 is excluded from the average. Sub-experiment f uses fewer tests and achieves full branch coverage on Tax. **randy** (sub-experiment e) does much better than sub-experiment g. Sub-experiment g could not cover Tax within *Gmax*. We attribute

randy's relatively good performance to the search space which is [defined to be] quite small and highly feasible; two conditions favorable to random search. Wider test intervals would prove more difficult for randy. That there is a considerable performance difference between sub-experiments f and g suggests the significant role selection pressure (**genet**) plays in covering the Tax program. Selection helps useful schema survive to the next generation.

Table 6.2 Results for Tax

	Number of branches & tests				
Seed	e		f		g
11	34	590	34	302	33
13	34	790	34	518	31
17	33	1050	34	194	33
19	34	210	34	212	33
23	34	250	34	446	28
Average	33.8	578	34	334.4	31.6
Excl. #17	34	460	34	369.5	31.25

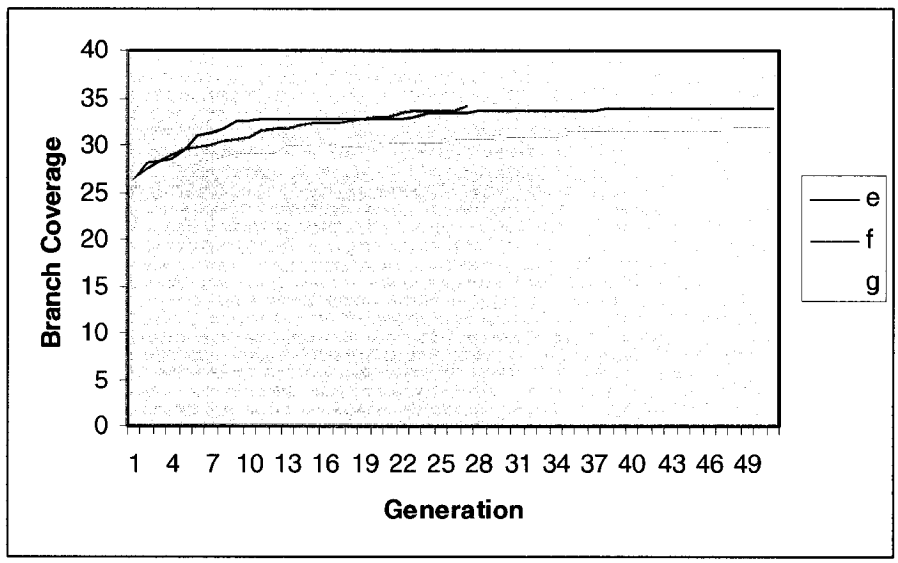


Figure 6.2 Evolutions of Tax sub-experiments

Figure 6.2 shows the coverage achieved as the search progresses through the generations. The steep ascent of the graph for sub-experiment f shows high evolvability. The search without selection pressure (sub-experiment g) performs even worse than

random search. Another reason for this poor result is the uniform crossover operator prevents chromosomes from changing too much, so the search is hindered from exploring the input space as aggressively as sub-experiment e.

Table 6.3 shows the tests from three runs in the order in which they were found by **genet** to cover predicates 14, 15 and 16. We analyzed these tests and found all but two, were produced by mutation which means selection and crossover helped to set the stage for useful mutation to occur. Mutation alone (sub-experiment e) was not as efficient as selection, crossover and mutation (sub-experiment f).

Table 6.3 Sample Tax test data

Seed		Test data	Trace	Origin
13	1	1 76 82 1 2221 1 0 0 0 495 1 1	F0 F2 F3 T4 T5 F8 T9 F12 F13 F14	I
13	2	1 61 47 1 996 1 0 0 1 474 0 1	F0 T2 T4 T5 F8 T9 F12 F13 T14 T15	M g0
13	3	1 80 47 1 996 1 0 0 1 309 0 1	F0 F2 T3 T4 T5 F8 T9 F12 F13 T14 F15 T16	M g20
13	4	1 80 48 1 996 1 0 0 1 309 0 0	F0 F2 T3 T4 T5 F8 T9 F12 F13 T14 F15 F16	M g25
17	5	1 84 50 1 2711 1 0 0 0 310 1 0	F0 F2 T3 T4 T5 F8 T9 F12 F13 F14	I
17	6	0 59 68 1 762 1 0 0 1 349 0 1	T0 T1 T4 T5 F8 T9 F12 F13 T14 F15 T16	I
17	7	0 59 68 1 762 1 0 0 1 476 0 1	T0 T1 T4 T5 F8 T9 F12 F13 T14 T15	M g0
17	8	1 84 73 1 3433 1 0 0 1 310 0 0	F0 F2 F3 T4 T5 F8 T9 F12 F13 T14 F15 F16	X g7
19	9	0 77 63 1 1367 1 0 0 0 471 0 1	T0 F1 T4 T5 F8 T9 F12 F13 F14	I
19	10	0 65 65 1 1023 1 0 0 1 471 0 1	T0 F1 T4 T5 F8 T9 F12 F13 T14 T15	X g0
19	11	0 65 65 1 1023 1 0 0 1 335 0 1	T0 F1 T4 T5 F8 T9 F12 F13 T14 F15 T16	M g1
19	12	0 65 65 1 1023 1 0 0 1 335 0 0	T0 F1 T4 T5 F8 T9 F12 F13 T14 F15 F16	M g4

I=initial population, X=crossover, M=mutation, gN=generation N

The similarity of gene value between test data 2 and 3 after a span of 20 generations illustrates schema survival. Notice the high resemblance between the following pairs of test data: 3 and 4, 6 and 7, and 11 and 12. The two crossovers occurred as follows:

before recombine 1 84 79 1 3400 1 0 0 0 310 0 0 | 1 79 73 1 3433 0 0 0 1 376 0 1

after recombine 1 79 79 1 3400 0 0 0 0 376 0 1 | **1 84 73 1 3433 1 0 0 1 310 0 0**

before recombine 0 58 65 1 1023 0 0 0 1 436 0 0 | 0 65 65 1 1409 1 1 0 0 471 1 1

after recombine 0 58 65 1 1409 0 1 0 0 436 1 0 | **0 65 65 1 1023 1 0 0 1 471 0 1**

6.3 Subalign

Subalign is a sequence alignment program written in C which is freely available (RDP 2001). The original (not instrumented) program is 2211 lines long[6]. Subalign takes 4 input strings: *Organism* filename, *GeneBank* filename, *Output* filename and *Options*; and we instrumented 44 of its predicates. These predicates deal with validating command line arguments.

Indirect real value representation is used: a Subalign chromosome is an array of 4 integers; each integer is an index into an array of real values. We created representative data for each input domain. The number of choices for *Organism*, *Genebank*, *Output* and *Options* are six, six, three and 30 respectively – these are minimal number of choices. This makes a total of 3,240 combinations. The number of choices for the input parameter *Option* is disproportionately higher than the rest. This suggests a higher mutation rate for the gene representing *Option* is appropriate.

Subalign is slightly different from Tax. It also deals with non-arithmetic predicates and its inputs do not share the same domain; but its chromosome is much shorter and the number of alleles per gene is higher. Like Tax, it is not evident what adaptation pattern is suitable to cover Subalign. Moreover unlike Triangle, biasing the genetic operators towards one test solution does not make it any easier to find other test solutions. Test solutions for Subalign are also necessarily similar, perhaps more so than Tax. This is because processing of the *Option* parameter depends on the first three parameters having no errors. Table 6.5 shows a few tests found by **genet** with random seed number 17.

[6] While experimenting with Subalign, we found an error in the program and corrected it. The variable ecoli on line 1987 should be initialized to NULL.

SubalignTestGenome makes use of uniform crossover and non-uniform mutation. The *Option* gene is seven times more likely to mutate than the other genes. The parameter values used in this experiment are: initpopSz = 50, parentSz = 10, Gmax = 50, Xr = 0.6, Mr = 0.8, Clf = 2 and Tmax = 0. Maximum number of tests is 1050. Three experiments were performed on Subalign:

h. **randy**,

i. **genet** with uniform crossover and non-uniform mutation, and

j. random parent selection (without **genet**), with uniform crossover and non-uniform mutation.

Table 6.4 reports the results of these sub-experiments. Sub-experiment i does much better than sub-experiments h and j. Sub-experiment i achieved full branch coverage for Subalign with an average of 587 tests. This result is encouraging for **genet** because the search space we defined for Subalign is small and feasible. Yet sub-experiment i which uses genet's fitness cum selection capability, outperforms randy. That sub-experiment i is performed significantly better than sub-experiment j illustrates the important of fitness based selection to the problem of covering the 44 predicates of Subalign.

Table 6.4 Results for Subalign

	Number of branches & tests			
Seed	h	i		j
11	85	88	860	78
13	84	88	554	86
17	86	88	356	87
19	83	88	842	81
23	86	88	320	83
Average	84.8	88	586.4	83

Figure 6.3 depicts that search progress of the Subalign experiments. The graph for sub-experiment i ascends more rapidly than the other two graphs; attesting to its superior quality of search.

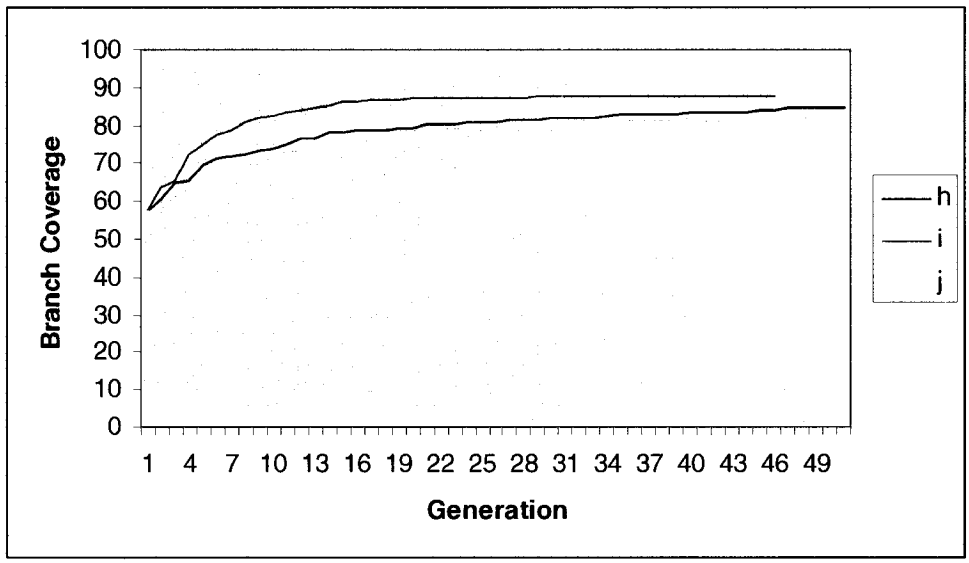


Figure 6.3 Evolutions of Subalign sub-experiments

Table 6.5 Sample Subalign test data

	Test data	Trace	Origin
1	0 3 0 15 1org_Ecoli.sid 3orgs_Ecoli_SSU_Prok.gb.good output1.seq -r=Mc.jannasc - p=666,672:734,740	F0 T7 F0 F7 T8 F0 F7 F8 T9 T0 F1 F2 F3 T4 F5 F6 T0 F1 F2 F3 F10 F12 T35 F35 F36 T35 F35 F36 T35 F35 T36 F16 F17 T18 T19 F20 F21 F22 F27 T32 F32 F22 T27 F28 F29 F30 F37 F38 F39 F40 F41 F37 F38 F39 F40 F41 F31 F42 F43 F33 T34 F34	I
2	3 3 0 1 2orgs_noEcoli.sid 3orgs_Ecoli_SSU_Prok.gb.good output1.seq	F0 T7 F0 F7 T8 F0 F7 F8 T9 F10 F12 T35 F35 T36 T35 F35 T36 T35 F35 F36 F16 F16 F17 F18 F19 F18 F19 T21 F33 T34 F34	I
3	2 1 0 13 2orgs_Ecoli.sid 2orgs_noEcoli_SSU_Prok.gb.good output1.seq -column=20,0	F0 T7 F0 F7 T8 F0 F7 F8 T9 T0 F1 F2 F3 F10 F12 T35 F35 F36 T35 F35 T36 F16 T16 F17 F18 F19 T21 T22 F23 F24 F25 F26 T33	I
4	2 3 0 15 2orgs_Ecoli.sid 3orgs_Ecoli_SSU_Prok.gb.good output1.seq -r=Mc.jannasc - p=666,672:734,740	F0 T7 F0 F7 T8 F0 F7 F8 T9 T0 F1 F2 F3 T4 F5 F6 T0 F1 F2 F3 F10 F12 T35 F35 F36 T35 F35 T36 T35 F35 T36 F16 F16 F17 F18 T19 T20 F22 F27 T32 F32 F22 T27 F28 F29 F30 F37 F38 F39 F40 F41 F37 F38 F39 F40 F41 F31 F42 F43 F33 T34 F34	X g1 before recombine 0 3 0 15 | 2 1 0 13 after recombine 0 1 0 13 | 2 3 0 15
5	2 3 0 17 2orgs_Ecoli.sid 3orgs_Ecoli_SSU_Prok.gb.good output1.seq -p=0,	F0 T7 F0 F7 T8 F0 F7 F8 T9 T0 F1 F2 F3 F10 F12 T35 F35 F36 T35 F35 T36 T35 F35 T36 F16 F16 F17 F18 F19 T18 F19 F21 F22 T27 F28 F29 F30 F37 F38 T39 T31	M g2 before mutate 2 3 0 15 after mutate 2 3 0 17
6	3 3 0 23 2orgs_noEcoli.sid 3orgs_Ecoli_SSU_Prok.gb.good output1.seq -p=670,572:734,740	F0 T7 F0 F7 T8 F0 F7 F8 T9 T0 F1 F2 F3 F10 F12 T35 F35 T36 T35 F35 T36 T35 F35 F36 F16 F16 F17 F18 F19 F18 F19 T21 F22 T27 F28 F29 F30 F37 T38 T31	M g3 before mutate 3 3 0 15 after mutate 3 3 0 23 3 3 0 15 produced by X in g1 before recombine 3 2 0 15 | 0 3 0 15 after recombine 3 3 0 15 | 0 2 0 15

6.4 Summary

1. The objective of the experiments was to ascertain the type of program that would benefit from using **genet**.
2. **genet**'s performances on three test programs: Triangle, Tax and Subalign were compared with that of **randy**'s, a comparable random test generator.
3. Table 6.6 compares the three test programs in five ways.

Table 6.6 A comparison of test programs

Test program (Branches)	**Triangle (20)**	**Tax (34)**	**Subalign (88)**
Solution density	Moderate	High	High
Decision type	Arithmetic	Non-arithmetic	Non-arithmetic
Chromosome length	3	12	4
Alleles per gene	High	Low	Low
Organization	Not necessary	Necessary	Necessary

4. The results of the experiments show that **genet** is more efficient than **randy** when it is necessary for tests to have information in common. This necessary organization of test solutions occurs in Tax and Subalign. As demonstrated by the with-and-without-**genet** sub-experiments, selection played a significant part in helping **genet** to outperform **randy** in Tax and Subalign.
5. Selection is less of a factor in Triangle because it is not necessary for its tests to be organized. Even without **genet** (but with appropriate adaptation operators), it was possible to outperform **randy**. However, the best performance was achieved with **genet** and with appropriate adaptation operators. An adaptation operator is considered appropriate if its use with **genet** outperforms **randy**.

Chapter 7 Conclusion and Suggestions for Further Work

The objective of this thesis was to find a simple way to automatically generate tests to cover the structure of a program. By simple, we mean to give minimal information to the automatic test generator (ATG). Consequently, the ATG is left with the burden of "learning" about the structure of a program and we suggest that this can be done quite effectively with the aid of formal concept analysis on trace data. This thesis proposes that a genetic algorithm and formal concept analysis is a simple and yet effective way to generate test for branch coverage automatically. **genet** is our implementation of this idea.

Ideally, one would like a systematic ATG such as **genet**, to at least outperform random search on any given test program. In reality, this is not possible according to the No Free Lunch theorem (Wolpert 1995). Therefore, we decided to make **genet** be composed of two parts: its fixed fitness cum selection core, untouchable to its users except through the parameters in the driver program and its variable adaptation mechanism, constructed by its users. Using this division, we could examine the interesting observation made in (Altenberg 1995) about the true source of power of genetic algorithms and the myth that schema abundance signals high search quality, a.k.a. evolvability, and find out whether genet's fitness cum selection core makes a difference in the search for test solutions. An alternative to our approach, and possibly more challenging, is to fix both the selection and adaptation mechanisms, and then search for types of programs where each configuration suits best.

The results of our experiments support the proposal of this thesis and they agree with present day theories (Wolpert 1995 and Altenberg 1995). Specifically, we found **genet** exerted more influence for Tax and Subalign than Triangle, and we conclude that

Tax and Subalign are representative of the types of programs well suited to **genet**. These two programs share the following characteristics: non-arithmetic decisions with a high degree of necessary organization amongst test solutions. Necessary organization means test solutions need to share information (Chaitin 1979) and so using selection pressure which is the force **genet** is essentially exerting to create fitter offspring makes sense as there is natural parent-offspring resemblance. After all, a genetic algorithm only works if there are patterns to be found (Mitchell 1996, page 118). Adaptation played a more significant role when the resemblance factor is less necessary.

One possible limitation with our evaluation of **genet** is that it does not compare **genet** with other ATGs in a direct way. This comparison could be useful to estimate the price, if any, of genet's "simplicity". A direct comparison may have been achieved for example by using the test programs experimented with by other ATGs or experimenting with other ATGs on our test programs. Triangle is one exception because its source code was listed in (McGraw 1998). We did however make some oblique comparisons. Their results illustrate **genet** exhibiting characteristics consistent with observations made of other ATGs (Khor 2004). A further criticism of our evaluation is the small number of test programs and their relatively small size. In software engineering where large presumably complex programs are deemed more interesting and carry more legitimacy, size does matter.

In addition to those already mentioned, there are a number of areas where **genet** could be improved or extended.

1. Although automatic test generation has been treated as a single optimum optimization problem previously (Korel 1990), we approached it as a

multimodal function or multiple constraint satisfaction problem and we are interested to find each optima rather than dwell on besting a solution. There is work relating to evolving dissimilar sub-populations or niches in a search space (Johnson 2001 provides some references) which may be interesting to explore to either enhance or replace genet's rather simplistic GA component. We did not go down this track in the present work to keep **genet** simple and relatively easy to analyze.

2. **genet** automatically associates tests it generates with the branches they exercise. We have suggested that this information could be useful for a number of software engineering tasks such as test selection, test minimization and understanding of dynamic program behavior (Khor 2004).

3. An incremental approach to concept formation (Godin 1995) could improve genet's execution time. We did not notice any significant difference between the execution times of **genet** and **randy** in our experiments.

4. Chapter 4 hinted at the misdirection problem which occurs when parent selection is continuously influenced by a particularly difficult (if not unreachable) branch and this may cause **genet** to under perform. **genet** tries to alleviate this problem with the *Tmax* parameter which is set by the user and counts the number of times a current target may not change before *Tmax* random individuals are introduced into a generation. This is a rather crude method. We did not use *Tmax* in our experiments (*Tmax*=0). But we did run **genet** with *Tmax* and got good results.

5. **genet** stops when it does not find any suitable parents. This situation and others having to do with unexpected errors could be handled more gracefully.

6. A more rigorous approach to the perennial problem of configuring GA parameters would be nice. We confined our experiments to an initial population size of 50 – 100 and used the same set of values for **genet**'s other parameters.

References

Agner Fog (2004). http://www.agner.org/random/randomc.htm

Altenberg, L. (1995) *The Schema theorem and Price's theorem.* In Foundations of Genetic Algorithms 3, ed. D. Whitley and M. Vose. Morgan Kauffman, San Francisco, pages 23-49.

Ball, T. (1999). *The concept of dynamic analysis.* ESEC/SIGSOFT FSE, pages 216-234, 1999.

Burgess, C. J. (2003). *Evolutionary Computing.* University of Bristol Computer Science Lecture Notes.

Cantu-Paz, E. (1997) *A survey of parallel genetic algorithms.* Illigal Report No. 97003, May, 1997.

Chaitin, G. J. (1979). *Toward a mathematical definition of "life".* In R. D. Levine and M. Tribus, The Maximum Entropy Formalism, MIT Press, 1979, pages 477-498.

Chang, K., Cross, J., Carlisle, W. and S. Liao. (1996). *A performance evaluation of heuristics-based test case generation methods for software branch coverage.* International Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 4, pages 585-608, 1996.

Clarke, L. (1976). *A system to generate test data and symbolically execute programs.* IEEE Trans. Software Eng., vol. SE-2, no.3, pages 215-222, Sept. 1976.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Ph.D. thesis, University of Michigan, Ann Arbor. In (Mitchell, 1996 page 175).

DeMillo, R.A., Lipton, R.J. and F.G. Sayward. (1978) *Hints on test data selection: Help for the practicing programmer.* IEEE Transactions on Computer, vol. 11, part 4, pages 34-31, April 1978.

Duran, J. W. and S. Ntafos. (1981). *A report on random testing.* In Proceedings of the 5th International Conference on Software Engineering, San Diego, California, pages 179 – 183, 1981.

(FCAHome) Formal Concept Analysis Homepage
http://www.upriss.org.uk/fca/fca.html

Ferguson, R. and B. Korel. (1996). *The chaining approach for software test data generation.* ACM Transactions on Software Engineering Methodology, vol. 5, no. 1,

pages 63-86, 1996.

Ganter, B. and R. Wille. (1999). *Formal concept analysis: mathematical foundations.* Springer, 1999.
Godin, R., Missaoui, R. and H. Alaoui. (1995). *Incremental concept formation algorithms based on Galois (concept) lattices.* Computation Intelligence, vol. 11, no.2, pages 246-267, 1995.

Goldberg, D.E. (1989). *Genetic algorithms in search, optimization and machine learning.* Reading, MA: Addison-Wesley, 1989.

Goldberg, D. E. and K. Deb. (1991). *A comparative analysis of selection schemes used in genetic algorithms.* In Foundations of Genetic Algorithms, San Mateo, California, USA. Morgan Kauffman Publishers, pages 69-93, 1991.

Gray, F. (1953). *Pulse code communication.* United States Patent Number 2,632,058. March 17, 1953.

Grefenstette, J. J. (1986). *Optimization of control parameters for genetic algorithms.* IEEE Transactions on Systems, Man and Cybernetics 16, no. 1, pages 122-128. In (Mitchell, 1996 page 176).

Grefenstette, J. J. and J. E. Baker (1989). *How genetic algorithms work: A critical look at implicit parallelism.* In Proceedings of the 3rd International Conference on Genetic Algorithms. Morgan Kauffman, June 1989.

Harman, M., Hierons, L., Baresel, A. and H. Sthamer. (2002) *Improving evolutionary testing by flag removal.* In Genetic and Evolutionary Computation Conference (GECCO), 2002.

Haupt, R. L. and S. E. Haupt. (1998). *Practical genetic algorithms.* Wiley-Interscience, 1998.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press. 1975.

Ince, D.C. (1987). *The automatic generation of test data.* The Computer Journal, vol. 30, no. 1, pages 63-69, 1987.

Johnson C. G. (2001). *Understanding complex systems through examples: a framework for qualitative example finding.* University of Kent at Canterbury. April 2001, pages 26 and 27.

Jones, B.F., Sthamer, H.-H. and D.E. Eyres. (1996) *Automatic structural testing using genetic algorithms.* Software Engineering Journal, pages: 299-306, September, 1996.

Khor, S. and P. Grogono. (2004). *Using a genetic algorithms and formal concept analysis to generate branch coverage test data automatically.* Accepted by The 19th IEEE International Conference on Automated Software Engineering (ASE 2004).

Korel, B. (1990). *Automated software test data generation.* IEEE Transactions on Software Engineering, vol. 16, no. 8, August 1990.

Linding, C. (1999). *A concept analysis framework.* http://sensei.ieec.uned.es/manuales/tkconcept/welcome.html

McGraw, G., Michael, C., and M. Schatz. (1998) *Generating software test data by evolution.* Technical Report RSTR-018-97-01, February 1998.

Mitchell, M (1996). *An introduction to genetic algorithms.* MIT Press, 1996.

Montalbano, M. (1974). *Decision tables.* Chicago: Science Research Associates, 1974.

Ould, M. A. (1991). *Testing – a challenge to method and tool developers.* Software Engineering Journal, 1991 6(2) pages 59-64.

Pargas, R.P., Harrold, M.J. and R.R. Peck. (1999) *Test-data generation using genetic algorithms.* Journal of Software Testing, Verification and Reliability, 1999.

Radcliffe, N.J. and P.D. Surry. (1995). *Fundamental limitations on search algorithms: evolutionary computing in perspective.* In Computer Science Today, pages 275-291, 1995.

(RDP 2001) http://rdp.cme.msu.edu/download/programs/Subalign/

Sthamer, H.-H. (1995). *The automatic generation of software test data using genetic algorithms.* Ph.D. Thesis, University of Glamorgan. November 1995.

Syswerda, G. (1991). *Schedule optimization using genetic algorithms.* In L. Davis (Ed.), Handbook of Genetic Algorithms, New York: Van Nostrand Reinhold, page 347. In (Haupt 1998, page 92).

Syswerda, G. (1989). *Uniform crossover in genetic algorithms.* In J.D. Schaffer (Ed.), Proc. Of the Third International Conference on Genetic Algorithms, Los Altos, CA: Morgan Kaufmann, pages 2-9. In (Haupt 1998, page 106).

Tracey, N.J. (1997). *Test-case data generation using optimization techniques.* – first year dphil report. Department of Computer Science, University of York, 1997. In (Tracey 1998)

Tracey, N., Clark, J. and K. Mander. (1998). *Automated program flaw finding using simulated annealing.* ACM SIGSOFT Proceedings of the 1998 International

Symposium on Software Testing and Analysis.

Wolpert, D. H. and W.G. Macready (1995). *No free lunch theorems for search.* Tech. Rep. SFI-TR-95-02-010, Santa Fe Institute, 6. February, 1995.

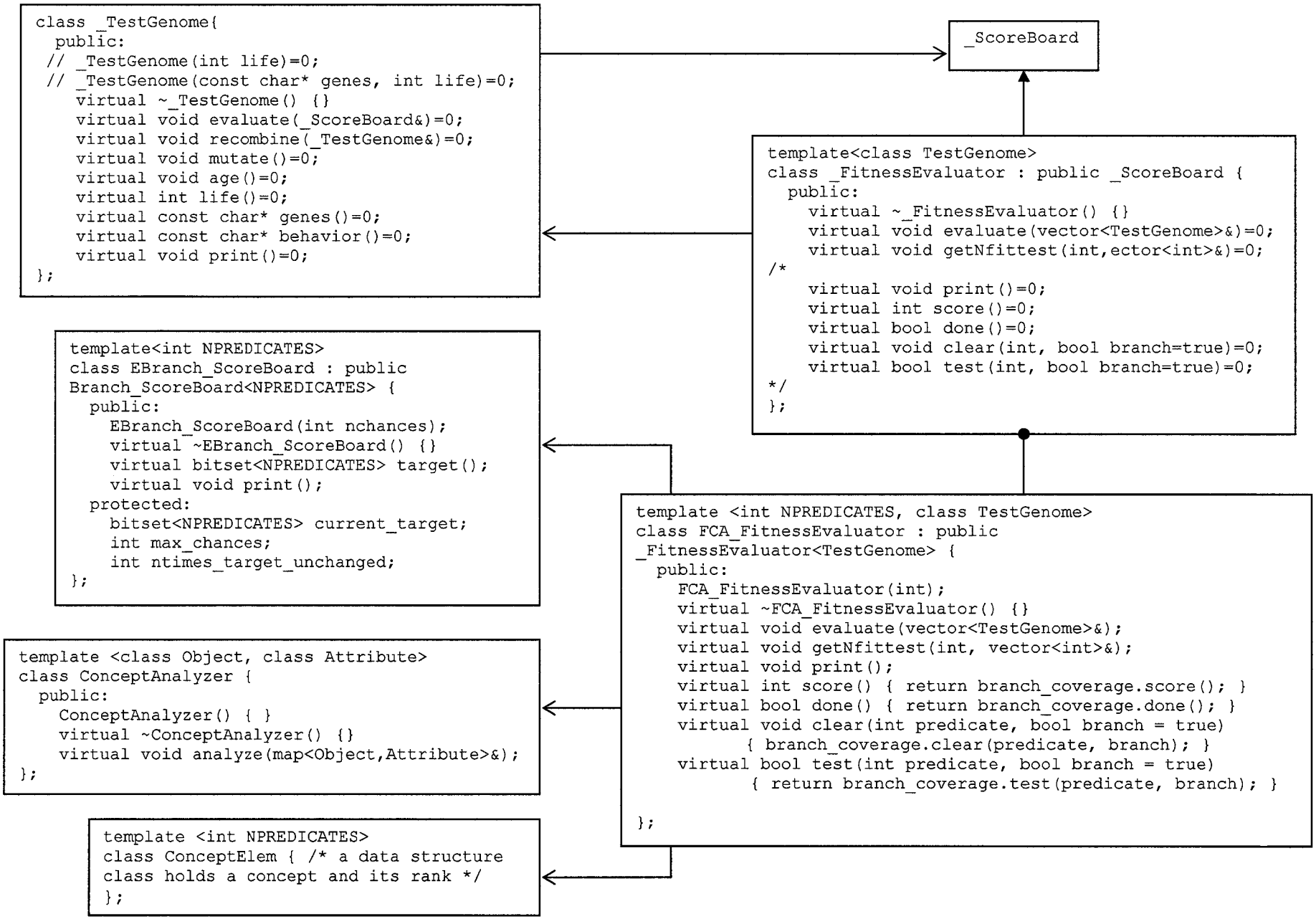
Appendix A1: Design and implementation of genet

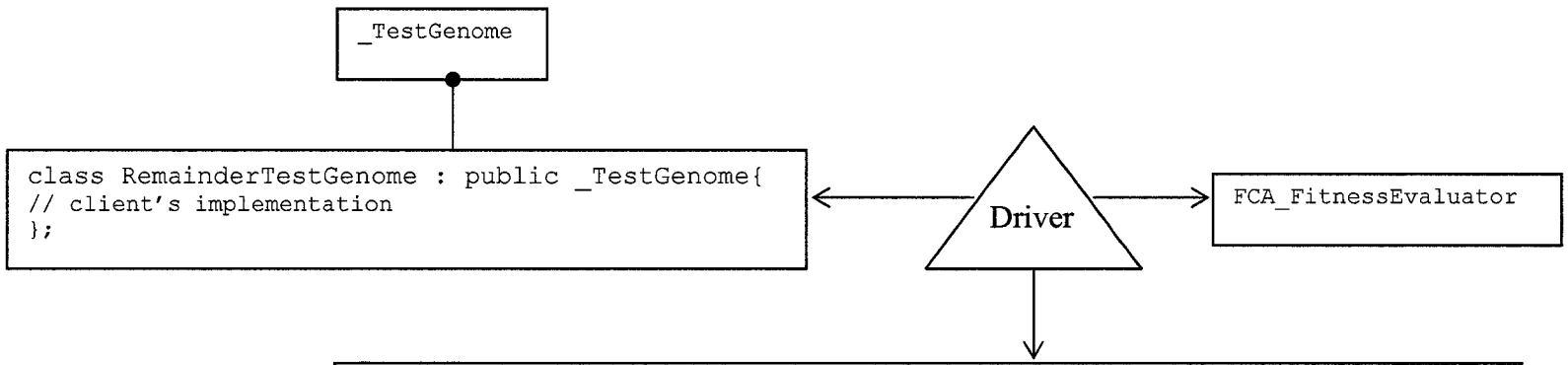
```
/* TriangleTestGenome.h */

#ifndef TRIANGLE_TEST_GENOME_H
#define TRIANGLE_TEST_GENOME_H

#include <vector>
#include <stdio.h>
#include "../../lib/_TestGenome.h"
#include "../../lib/randomc/randomc.h"
using namespace std;

const int DATA_SZ = 100;
const int TRACE_SZ = 100;
const int CMD_SZ = 120;

class TriangleTestGenome : public _TestGenome {
  public:
    TriangleTestGenome(int);
    TriangleTestGenome(const char*, int);
    virtual ~TriangleTestGenome() {}
    bool evaluate(_ScoreBoard&);
    void recombine(_TestGenome&) {}
    void recombine(TriangleTestGenome&);
    void mutate();
    void age() { --life_d; }
    void die() { life_d = 0; }
    int life() { return life_d; }
    void refresh_data();
    const char* genes() { return data_d; }
    const char* behavior() { return trace_d; }
    void print();
  private:
    inline int gene(int loci) { return genotype[loci]; }
    inline void gene(int loci, int val) { genotype[loci] = val; }
    static TRandomMersenne rmg;
    static int RandomInt(int min, int max)
         { return rmg.IRandom(min, max); }
    static int RandomGene() { return RandomInt(-1, 998); }
    void run_program();
    bool register_trace(_ScoreBoard&);
    char data_d[DATA_SZ + 1];
    char trace_d[TRACE_SZ + 1];
    int life_d;
    vector<int> genotype;
};

TRandomMersenne TriangleTestGenome::rmg(23);

TriangleTestGenome::TriangleTestGenome(int life) {
  life_d = life;
  genotype.push_back(RandomGene());
  genotype.push_back(RandomGene());
  genotype.push_back(RandomGene());
```

```
  refresh_data();
}

void TriangleTestGenome::refresh_data() {
  sprintf(data_d, "%d %d %d", genotype[0], genotype[1], genotype[2]);
}

TriangleTestGenome::TriangleTestGenome(const char* input_s, int life) {
  life_d = life;
  snprintf(data_d, DATA_SZ+1, "%s", input_s);
  vectorize(input_s, genotype);
}

bool TriangleTestGenome::evaluate(_ScoreBoard& fe) {
  run_program(); // writes to trace.out
  ifstream trfin("trace.out");
  assert(trfin); // file opened ok
  trfin.getline(trace_d, TRACE_SZ);
  assert(trfin); // line read ok
  trfin.close();
  return (register_trace(fe));
}

void TriangleTestGenome::print() {
  cout << "data  " << data_d << endl;
  cout << "trace " << trace_d << endl;
  cout << "life  " << life_d << endl;
}

// writes to trace.out
void TriangleTestGenome::run_program() {
  char cmd[CMD_SZ + 1];
  snprintf(cmd, CMD_SZ, "%s %s", "java Triangle ", data_d);
  system(cmd);
}

bool TriangleTestGenome::register_trace(_ScoreBoard& fe) {
  char *ts = new char[strlen(trace_d) + 1];
  assert(ts != 0);
  strcpy(ts, trace_d);
  char *p = strtok(ts, " ");
  int pos = 0;
  bool changed = false;
  while (p != NULL) {
    assert((p[0] == 'T') || (p[0] == 'F'));
    if (p[0] == 'T') {
      pos = atoi(p+1);
      if (fe.test(pos, true)) {
        fe.clear(pos, true);
        changed = true;
      }
    }
    else {  // (p[0] == 'F')
      pos = atoi(p+1);
      if (fe.test(pos, false)) {
        fe.clear(pos, false);
        changed = true;
```

```
      }
    }
    p = strtok(NULL, " ");
  }
  delete [] ts;
  return changed;
}

void TriangleTestGenome::recombine(TriangleTestGenome& tg) {
// shuffle crossover
  int xloci1 = RandomInt(0,2); int xloci2 = RandomInt(0,2);
  int tmp = 0;
  tmp = genotype[xloci1];
  genotype[xloci1] = tg.gene(xloci2);
  tg.gene(xloci2, tmp);

// uniform crossover
/*  int xloci = RandomInt(0,1);
  int tmp = 0;
  for (int i = xloci; i < genotype.size(); i+=2) {
    tmp = genotype[i];
    genotype[i] = tg.gene(i);
    tg.gene(i, tmp);
  }
*/
  refresh_data();
  tg.refresh_data();
}

// uniform mutation
void TriangleTestGenome::mutate() {
  int mloci = RandomInt(0, genotype.size()-1);
  genotype[mloci] = RandomGene();
  refresh_data();
}

#endif
```

```
/* TriangleEvolTestDriver.C */

#include "../../lib/EvolDATG.h"
#include "../../lib/FCA_FitnessEvaluator.h"
#include "TriangleTestGenome.h"

int main(int argc, char *argv[]) {
  // srand(time(NULL)); // for experiments without genet
  int num_parents_iter = 10;
  //  int num_predicates = 10;
  int num_iters = 50;
  char inputfile[] = "Triangle.testdata";
  int input_line_size = 20;
  double pcrossover = 0.6;
  double pmutation = 0.8;
  int glife = 2;
  int num_chances = 0;
  int init_popsize = 100;
  EvolDATG<TriangleTestGenome,
     FCA_FitnessEvaluator<10,TriangleTestGenome> >
     edatg(num_parents_iter, num_iters, inputfile, input_line_size,
     pcrossover, pmutation, glife, num_chances, init_popsize);
  long begin = time(NULL);
  edatg.run();
  long end = time(NULL);
  edatg.print();
  printf("\n%s %ld\n", "Execution time: ", (end - begin));
  exit(0);
}
```


class _TestGenome{
public:
// _TestGenome(int life)=0;
// _TestGenome(const char* genes, int life)=0;
virtual ~_TestGenome() {}
virtual void evaluate(_ScoreBoard&)=0;
virtual void recombine(_TestGenome&)=0;
virtual void mutate()=0;
virtual void age()=0;
virtual int life()=0;
virtual const char* genes()=0;
virtual const char* behavior()=0;
virtual void print()=0;
};
_ScoreBoard
template<class TestGenome>
class _FitnessEvaluator : public _ScoreBoard {
public:
virtual ~_FitnessEvaluator() {}
virtual void evaluate(vector<TestGenome>&)=0;
virtual void getNfittest(int,ector<int>&)=0;
/*
virtual void print()=0;
virtual int score()=0;
virtual bool done()=0;
virtual void clear(int, bool branch=true)=0;
virtual bool test(int, bool branch=true)=0;
*/
};
template<int NPREDICATES>
class EBranch_ScoreBoard : public
Branch_ScoreBoard<NPREDICATES> {
public:
EBranch_ScoreBoard(int nchances);
virtual ~EBranch_ScoreBoard() {}
virtual bitset<NPREDICATES> target();
virtual void print();
protected:
bitset<NPREDICATES> current_target;
int max_chances;
int ntimes_target_unchanged;
};
template <int NPREDICATES, class TestGenome>
class FCA_FitnessEvaluator : public
_FitnessEvaluator<TestGenome> {
public:
FCA_FitnessEvaluator(int);
virtual ~FCA_FitnessEvaluator() {}
virtual void evaluate(vector<TestGenome>&);
virtual void getNfittest(int, vector<int>&);
virtual void print();
virtual int score() { return branch_coverage.score(); }
virtual bool done() { return branch_coverage.done(); }
virtual void clear(int predicate, bool branch = true)
{ branch_coverage.clear(predicate, branch); }
virtual bool test(int predicate, bool branch = true)
{ return branch_coverage.test(predicate, branch); }
};
template <class Object, class Attribute>
class ConceptAnalyzer {
public:
ConceptAnalyzer() { }
virtual ~ConceptAnalyzer() {}
virtual void analyze(map<Object,Attribute>&);
};
template <int NPREDICATES>
class ConceptElem { /* a data structure
class holds a concept and its rank */
};




```
template<class TestGenome, class FitnessEvaluator>
class EvolDATG {
  public:
    EvolDATG(int, int, const char*, int, double, double, int, int, int);
    virtual ~EvolDATG() { delete [] testfile; }
    void run();
    void print();
    int current_iteration() { return current_iteration_d; }
    int score() { return fitness_evaluator.score(); }
  protected:
    void initialize_population();
    void evolve_next_generation();
    int select_parent();
    void mutate();
    void recombine();
    void introduce();
    int ntests_per_iteration_d;  // population size
    int max_iterations_d;        // max generations
    int current_iteration_d;
    char* testfile; int input_sz;
    double pcrossover; double pmutation;
    int life_d; int max_tries; int gen0_sz;
    bool introduce_new_individuals;
    vector<TestGenome> all_tests;
    vector<TestGenome> new_tests;
    vector<TestGenome> min_tests;
    FitnessEvaluator fitness_evaluator;
    vector<int> parents;
};
```

Appendix A2: Design and implementation of randy

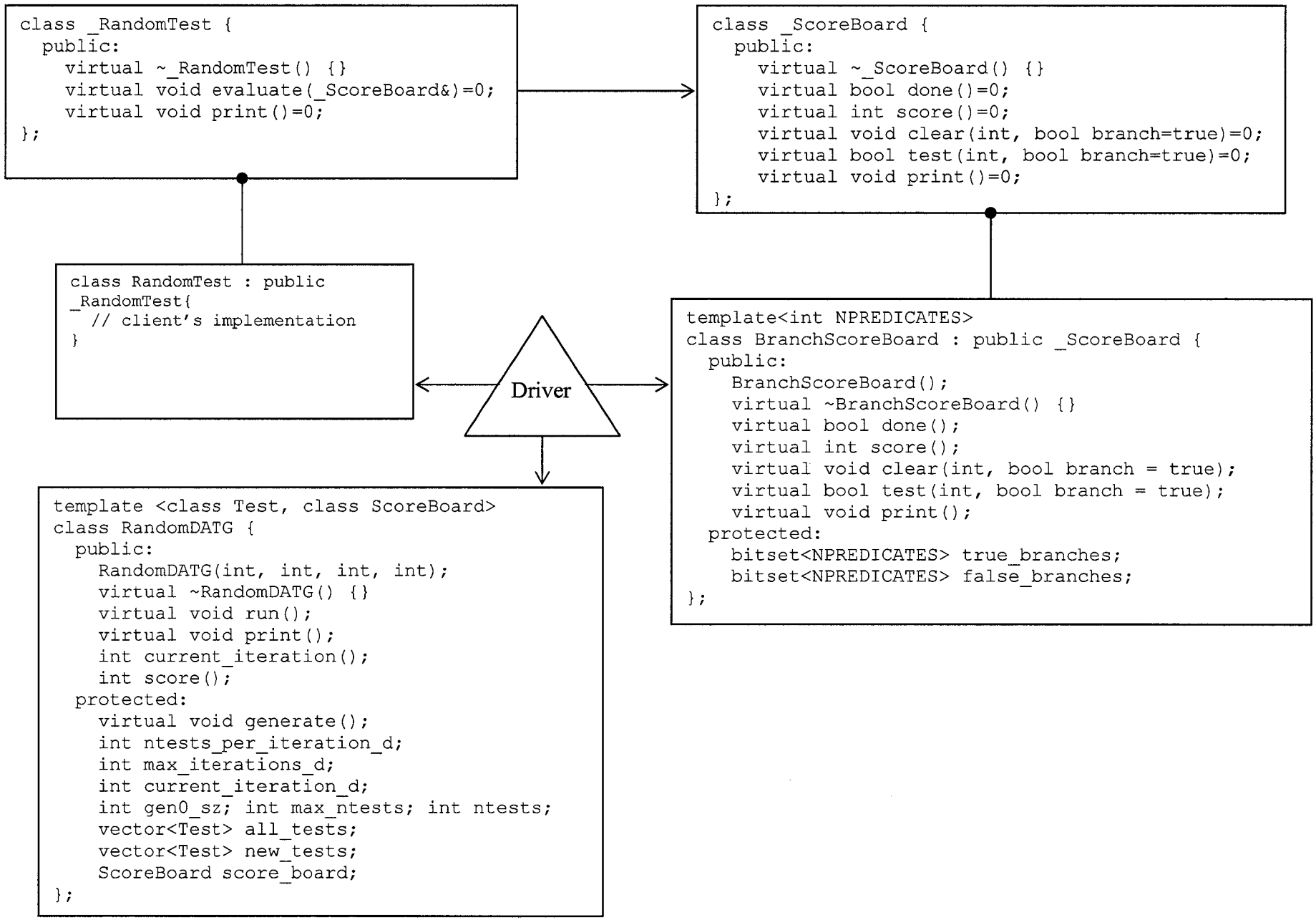
/* TriangleRandomTest.h*/

```
#ifndef TRIANGLE_RANDOM_TEST
#define TRIANGLE_RANDOM_TEST

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "../../lib/_RandomTest.h"
#include "../../lib/randomc/randomc.h"
using namespace std;

const int DATA_SZ = 100;
const int TRACE_SZ = 100;
const int CMD_SZ = 120;

class TriangleRandomTest : public _RandomTest {
  public:
    TriangleRandomTest();
    virtual ~TriangleRandomTest() {}
    virtual void evaluate(_ScoreBoard&);
    virtual void print();
  private:
    void run_program();
    void report_score(_ScoreBoard&);
    char data[DATA_SZ + 1];
    char trace[TRACE_SZ + 1];

    static TRandomMersenne rmg;
    static int RandomInt(int min, int max)
              { return rmg.IRandom(min, max); }
};

TRandomMersenne TriangleRandomTest::rmg(23);

TriangleRandomTest::TriangleRandomTest() {
  int a = RandomInt(-1, 998);
  int b = RandomInt(-1, 998);
  int c = RandomInt(-1, 998);
  sprintf(data, "%d %d %d", a, b, c);
}

void TriangleRandomTest::evaluate(_ScoreBoard& sb) {
  run_program(); // writes to trace.out
  ifstream trfin("trace.out");
  assert(trfin); // file opened ok
  trfin.getline(trace, TRACE_SZ);
  assert(trfin); // line read ok
  trfin.close();
  report_score(sb);
}
```

```
void TriangleRandomTest::print() {
  cout << "data  " << data << endl;
  cout << "trace " << trace << endl;
}

// writes to trace.out
void TriangleRandomTest::run_program() {
  char cmd[CMD_SZ + 1];
  sprintf(cmd,"%s %s", "java Triangle", data);
  system(cmd);
}

// read from trace.out
// trace example: "T0 F1 F500"
void TriangleRandomTest::report_score(_ScoreBoard& sb) {
  char *ts = new char[strlen(trace) + 1];
  assert(ts != 0);
  strcpy(ts, trace);
  char *p = strtok(ts, " ");
  while (p != NULL) {
    if (p[0] == 'T')
      sb.clear(atoi(p+1), true);
    else if (p[0] == 'F')
          sb.clear(atoi(p+1), false);
        else
          cerr << "Something is wrong with " << trace << endl;
    p = strtok(NULL, " ");
  }
  delete [] ts;
}

#endif
```

/* TriangleRandomTestDriver.C */

```
#include "../../lib/RandomDATG.h"
#include "../../lib/Branch_ScoreBoard.h"
#include "TriangleRandomTest.h"

int main(int argc, char* argv[]) {
  int num_tests_iter = 20;
  int num_iters = 50;
  int init_popsz = 100;
  int max_tests = 1100;

  RandomDATG<TriangleRandomTest, Branch_ScoreBoard<10> >
         rdatg(num_tests_iter, num_iters, init_popsz, max_tests);
  long begin = time(NULL);
  rdatg.run();
  long end = time(NULL);
  rdatg.print();
  printf("\n%s %ld\n", "Execution time: ", (end - begin));
  exit(0);
}
```


class _RandomTest {
public:
virtual ~_RandomTest() {}
virtual void evaluate(_ScoreBoard&)=0;
virtual void print()=0;
};
class _ScoreBoard {
public:
virtual ~_ScoreBoard() {}
virtual bool done()=0;
virtual int score()=0;
virtual void clear(int, bool branch=true)=0;
virtual bool test(int, bool branch=true)=0;
virtual void print()=0;
};
class RandomTest : public
_RandomTest{
// client's implementation
}
Driver
template<int NPREDICATES>
class BranchScoreBoard : public _ScoreBoard {
public:
BranchScoreBoard();
virtual ~BranchScoreBoard() {}
virtual bool done();
virtual int score();
virtual void clear(int, bool branch = true);
virtual bool test(int, bool branch = true);
virtual void print();
protected:
bitset<NPREDICATES> true_branches;
bitset<NPREDICATES> false_branches;
};
template <class Test, class ScoreBoard>
class RandomDATG {
public:
RandomDATG(int, int, int, int);
virtual ~RandomDATG() {}
virtual void run();
virtual void print();
int current_iteration();
int score();
protected:
virtual void generate();
int ntests_per_iteration_d;
int max_iterations_d;
int current_iteration_d;
int gen0_sz; int max_ntests; int ntests;
vector<Test> all_tests;
vector<Test> new_tests;
ScoreBoard score_board;
};

Appendix B: Test programs

/* Remainder.java */

```
import java.io.*;

public class Remainder {
  private static FileWriter fout;

  private static void remainder(int A, int B) throws IOException {
    int R = -1;
    int Cy = 0, Ny = 0;
    if (A == 0) {
      fout.write("T0 ");
    }
    else {
      fout.write("F0 ");
      if (B == 0) {
        fout.write("T1 ");
      }
      else {
        fout.write("F1 ");
        if (A > 0) {
          fout.write("T2 ");
          if (B > 0) {
            fout.write("T3 ");
            while ((A - Ny) >= B) {
           Ny = Ny + B;
           R = A - Ny;
           Cy = Cy + 1;
         }
          }
       else {
            fout.write("F3 ");
         while ((A + Ny) >= java.lang.Math.abs(B)) {
           Ny = Ny + B;
           R = A + Ny;
           Cy = Cy - 1;
         }
          }
        }
     else {
          fout.write("F2 ");
          if (B > 0) {
            fout.write("T4 ");
            while (java.lang.Math.abs(A + Ny) >= B) {
              Ny = Ny + B;
           R = A + Ny;
           Cy = Cy - 1;
         }
          }
       else {
            fout.write("F4 ");
         while ((A - Ny) <= B) {
           Ny = Ny + B;
```

```
            R = java.lang.Math.abs(A - Ny);
            Cy = Cy + 1;
          }
           }
         }
       }
     }
     System.out.println(A + "/" + B + " = " + Cy + " " + R);
   }

   public static void main(String[] args) throws IOException {
     fout = new FileWriter("trace.out");
     int n = Integer.parseInt(args[0]);
     int d = Integer.parseInt(args[1]);
     remainder(n, d);
     fout.close();
   }
}
```

/* Triangle.java */

```
import java.io.*;
import java.lang.*;

public class Triangle {
  private static String[] TRIANGLE_TYPES = {" ", "scalene",
          "isosceles", "equilateral", "not a triangle"};
  private static FileWriter fout;

  public static void main(String[] args) throws IOException {
    fout = new FileWriter(new File("trace.out"));
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    int c = Integer.parseInt(args[2]);
    int d = triang(a, b, c);
    System.out.println(Integer.toString(a) + " " + Integer.toString(b)
         + " " + Integer.toString(c) + " " + TRIANGLE_TYPES[d]);
    fout.close();
  }

  private static int triang(int a, int b, int c) throws IOException {
    if ((a <= 0) || (b <= 0) || (c <= 0)) {
      fout.write("T0 ");
      return 4;
    }
    else
      fout.write("F0 ");

    int tri = 0;

    if (a == b) {
      fout.write("T1 ");
      tri += 1;
    }
    else
      fout.write("F1 ");

    if (a == c) {
      fout.write("T2 ");
      tri += 2;
    }
    else
      fout.write("F2 ");

    if (b == c) {
      fout.write("T3 ");
      tri += 3;
    }
    else
      fout.write("F3 ");

    if (tri == 0) {
      fout.write("T4 ");
      if ((a + b <= c) || (b + c <= a) || (a + c <= b)) {
        fout.write("T5 ");
```

```
        tri = 4;
      }
      else {
        fout.write("F5 ");
     tri = 1;
      }
      return tri;
    }
    else
      fout.write("F4 ");

    if (tri > 3) {
      fout.write("T6 ");
      tri = 3;
    }
    else {
      fout.write("F6 ");
      if ((tri == 1) && (a + b > c)) {
        fout.write("T7 ");
        tri = 2;
      }
      else {
        fout.write("F7 ");
        if ((tri == 2) && (a + c > b)) {
          fout.write("T8 ");
          tri = 2;
        }
        else {
          fout.write("F8 ");
          if ((tri == 3) && (b + c > a)) {
            fout.write("T9 ");
            tri = 2;
          }
          else {
            fout.write("F9 ");
            tri = 4;
          }
        }
      }
    }
    return tri;
  }
}
```

/* Tax.java */

```
import java.io.*;
import java.lang.*;

public class Tax {
  private static String[] TaxStatus = {"Obtain publication 519.",
              "File a tax return.", "Do not file a tax return.",
              "File for a refund."};
  private static FileWriter fout;

  public static void main(String[] args) throws IOException {
    fout = new FileWriter(new File("trace.out"));
    // marital status; 0=single, 1=married
    int a = Integer.parseInt(args[0]);
    // age; 45 - 85
    int b = Integer.parseInt(args[1]);
    // spouse age; 45 - 85
    int c = Integer.parseInt(args[2]);
    // 0=not citizen or resident, 1=citizen or resident
    int d = Integer.parseInt(args[3]);
    // gross income; 400 - 800
    int e = Integer.parseInt(args[4]);
    // married at end of year; 0=no, 1=yes
    int f = Integer.parseInt(args[5]);
    // other person claiming; 0=no, 1=yes
    int g = Integer.parseInt(args[6]);
    // filing separate returns; 0=no, 1=yes
    int h = Integer.parseInt(args[7]);
    // living together at end of last year; 0=no, 1=yes
    int i = Integer.parseInt(args[8]);
    // self-employed income; 300 - 500
    int j = Integer.parseInt(args[9]);
    // tips; 0=no, 1=yes
    int k = Integer.parseInt(args[10]);
    // witheld tax; 0=no, 1=yes
    int l = Integer.parseInt(args[11]);

    int ts = tax_return(a, b, c, d, e, f, g, h, i, j, k, l);
    for(int n = 0; n < args.length; n++)
      System.out.print(args[n] + " ");
    System.out.println(TaxStatus[ts]);
    fout.close();
  }

  private static int tax_return(int a, int b, int c, int d, int e,
          int f, int g, int h, int i, int j, int k, int l)
          throws IOException {
    int mtgi = 0;       // minimum taxable income
    if (a == 0) {       // single
      fout.write("T0 ");
      if (b < 65) {     // age
        fout.write("T1 ");
        mtgi = 1700;
      }
      else {            // age >= 65
```

```
    fout.write("F1 ");
    mtgi = 2300;
  }
}
else {              // (a == 1) married
  fout.write("F0 ");
  if (b < 65 && c < 65) {
    fout.write("T2 ");
    mtgi = 2300;
  }
  else {
   fout.write("F2 ");
    if (b < 65 || c < 65) {
      fout.write("T3 ");
     mtgi = 2900;
    }
    else {          // (b >= 65 && c>= 65)
      fout.write("F3 ");
      mtgi = 3500;
    }
  }
}
int ts = 0;         // tax status
if (d == 0) {       // not citizen or resident
  fout.write("F4 ");
  ts = 0;
}
else {              // citizen or resident
  fout.write("T4 ");
  if (e <= 600) {   // gross income
   fout.write("F5 ");
    if ((j > 400) || (k == 1)) { // self employed income or tips
      fout.write("T6 ");
      ts = 1;
    }
    else {
      fout.write("F6 ");
      if (l == 1) { // witheld tax
        fout.write("T7 ");
        ts = 2;
      }
      else {
        fout.write("F7 ");
        ts = 3;
      }
    }
  }
  else {       // gross income > 600
   fout.write("T5 ");
   if (e > mtgi) {
      fout.write("T8 ");
      ts = 2;
    }
    else {
     fout.write("F8 ");
     if (f == 0) {  // not married at end of last year
       fout.write("F9 ");
```

```
          if ((k == 1) || (j > 400)) {
            fout.write("T10 ");
            ts = 1;
             }
          else {
            fout.write("F10 ");
            if (l == 1) {
              fout.write("T11 ");
              ts = 2;
            }
            else {
              fout.write("F11 ");
              ts = 3;
            }
          }
         }
         else {
           fout.write("T9 ");
          if (g == 1) {
               fout.write("T12 ");
            ts = 1;
             }
             else {
            fout.write("F12 ");
            if (h == 1) {
              fout.write("T13 ");
              ts = 1;
               }
            else {
              fout.write("F13 ");
              if (i == 0) {
               fout.write("F14 ");
               ts = 1;
              }
              else {
               fout.write("T14 ");
               if ((k == 1) || (j > 400)) {
                 fout.write("T15 ");
                 ts = 1;
               }
               else {
                 fout.write("F15 ");
                 if (l == 1) {
                   fout.write("T16 ");
                   ts = 2;
                 }
                 else {
                   fout.write("F16 ");
                   ts = 3;
                 }
               }
             }
            } } } } }
    }
    return ts;
  }
}
```

/* Subalign.c: only instrumented portion of program is listed. */

```
/*
Copyright (c) 1991-1992, University of Illinois board of trustees.  All
rights reserved.  Written by Michael Maciukenas at the Ribosomal
Database Project. Design and implementation guidance by Niels Larsen,
Gary Olsen, Carl Woese.
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>

FILE *trace;
char tracefile[] = "trace.out";


...

getentries(orgs, gen, log)
/* store the entries from gen (genbank) in the orgs list */
/* write log info to log */
list orgs;
FILE *gen, *log;
{
     char *l;
     list lines;        /* lines for the current genbank entry */
     int requested_entry;    /* =1 if it's still possible that this
                             entry was requested */
     organism taxa;          /* organism that the entry matches */
     char *sequence;    /* temporary space for reading in sequence */
     char name[MAXLEN];
     int len;
     int i;
     int ch;
     int data_line;
     int dummy;
     int more;

    while((l=getlin(gen, &ch))!=NULL)
    {
          /* have first line in entry */
          lines=newlist();
          requested_entry=1;
          taxa=NULL;
          sequence=NULL;
          more=1;

          while(more)
          {
               if(l==NULL)
               {
                    fprintf(stderr,
                         "ERROR--Unexpected EOF in genbank file\n");
                    fclose(trace);
                    exit(0);
               }
```

```
                else if(compare(l, ENDENTRY))
                     more=0;
                else if(requested_entry)
                {
                     if(!addnode(lines, l))
                     {
                     /* not enough memory to create add line to entry */
                          fprintf(stderr,
                     "ERROR--Not enough memory to read genbank file\n");
                          fclose(trace);
                          exit(0);
                     }
                     if(compare(l, LOCUSLINE))
                     {
                          /* find if name is in orgs */
                          i=sscanf(l, "LOCUS %s BP", name);
                          if(i==0 || i==EOF)
                          {
                               fprintf(trace, "%s", "T35 ");
                               /* badly formatted LOCUS line */
                               /* so we will ignore this entry */
                               requested_entry=0;
                          }
                          else
                          {
/*Susan: T35 'cos couldn't figure out how to make a bad LOCUS line */
                               fprintf(trace, "%s", "T35 F35 ");
                               if((taxa=find_org(orgs, name))!=NULL)
                               {
                                    fprintf(trace, "%s", "T36 ");
                               /* organism was requested */
                               /* now taxa=the organism for this entry */
                                    ;
                               }
                               else
                               {
                                    fprintf(trace, "%s", "F36 ");
                                    /* organism was not requested */
                                    requested_entry=0;
                               }
                          }
                          /* get line after locus line */
                          l=getlin(gen, &ch);
                     }
                     else if(compare(l, ORIGINLINE))
                     {
                          if(sequence==NULL)
                          {
                          /* read the sequence after the origin line */
                               data_line=1;
                     sequence=getsequence(gen, log, &len, &l, &data_line);
/* if l is returned, it's the line in gen, so no need to getlin() */
                               if(l==NULL)
                                    l=getlin(gen, &ch);
                               /* store length */
                               taxa->len=len;
                          }
```

```
                    else
                    {
               /* more than one sequence per this entry, so ignore */
                    char *tmp=getsequence(gen, log, &len, &l, &dummy);
                         if(tmp!=NULL)
                              free(tmp);
  /* if l is returned, it's the line in gen, so no need to getlin() */
                         if(l==NULL)
                              l=getlin(gen, &ch);
                    }
               }
               else
                    /* get next line */
                    l=getlin(gen, &ch);
          }
          else
               /* get next line if not requested_entry */
               l=getlin(gen, &ch);
     }
     if(requested_entry && taxa!=NULL)
     {
          /* setup taxa */
          taxa->genbank_lines=lines;
          taxa->sequence=sequence;
          taxa->data_line=data_line;
     }
     else
     {
          /* throw away lines that were read in */
          free_line_list(lines);
          /* throw away the sequence if it was read */
          if(sequence!=NULL)
               free(sequence);
     }
  }
  /* now we've read in all the entries */
  /* so we can return */
}

...

int read_pair_list(s, starts, ends, astart, aend, errorstring)
char *s;
list starts, ends;
int astart, aend;
char *errorstring;
{
     int i, j, ss, ee, os, oe, num, error;

     startlist(starts);
     startlist(ends);

     os= -1;
     oe= -1;
     num=1;
```

```
for(;*s!='\0';)
{
     if(num!=1)
          s++;
     error=read_pair(s, &ss, &ee, astart, aend);
     if(error)
     {
          fprintf(trace, "%s", "T37 ");
          fprintf(stderr, "ERROR--%s: Error reading Range %d\n",
               errorstring, num);
          return(1);
     }
     fprintf(trace, "%s", "F37 ");
     if(ee<ss)
     {
          fprintf(trace, "%s", "T38 ");
          fprintf(stderr,
                    "ERROR--%s: End before Start in Range %d\n",
               errorstring, num);
          return(1);
     }
     else
     {
          fprintf(trace, "%s", "F38 ");
          if(ss<=0)
          {
               fprintf(trace, "%s", "T39 ");
               fprintf(stderr,
                    "ERROR--%s: Range %d starts at less than 1\n",
                    errorstring, num);
               return(1);
          }
          else
          {
               fprintf(trace, "%s", "F39 ");
               if(ss<os)
               {
                    fprintf(trace, "%s", "T40 ");
                    fprintf(stderr,
                    "ERROR--%s: Range %d covers other ranges\n",
                         errorstring, num);
                    return(1);
               }
               else
               {
                    fprintf(trace, "%s", "F40 ");
                    if(ss<=oe)
                    {
                         fprintf(trace, "%s", "T41 ");
                         fprintf(stderr,
                         "ERROR--%s: Ranges %d and %d overlap\n",
                         errorstring, num-1, num);
                         return(1);
                    }
                    else
                    {
                         fprintf(trace, "%s", "F41 ");
```

```
                                addnode(starts, ss);
                                addnode(ends, ee);
                           }
                      }
                 }
            }
            os=ss;
            oe=ee;
            num++;
            while(*s!=':' && *s!='\0')
                 s++;
       }
       return(0);
}

...

int calc_seq_cols(seq_starts, seq_ends, seq_col_starts, seq_col_ends,
     reference)
/* updates seq_col_starts,ends */
/* returns 0 if error found in sequence positions */
list seq_starts, seq_ends, seq_col_starts, seq_col_ends;
organism reference;
{
     char *s;
     list start, end;
     int ss, ee;
     int curbase, curcol;
     int first;
     int need_end;
     int range_num;

     curbase=0;
     curcol= -1;
     start=firstnode(seq_starts);
     end=firstnode(seq_ends);
     startlist(seq_col_starts);
     startlist(seq_col_ends);
     need_end=0;
     range_num=1;
     if(start!=NULL && end!=NULL)
     {
          for(s=reference->sequence;*s!='\0';s++)
          {
               curcol++;
               if(base(*s))
                    curbase++;
               if(start!=NULL && curbase==thisint(start))
               {
                    addnode(seq_col_starts, curcol);
                    need_end=1;
                    start=nextnode(start);
               }
               if(end!=NULL && curbase==thisint(end))
               {
                    addnode(seq_col_ends, curcol);
```

```
                    need_end=0;
                    end=nextnode(end);
                    range_num++;
               }
          }
          if(need_end)
          {
               fprintf(trace, "%s", "T42 ");
               fprintf(stderr,
               "ERROR--Range %d extends past end of reference
               sequence\n", range_num);
               return(0);
          }
          else
          {
               fprintf(trace, "%s", "F42 ");
               if(start!=NULL)
               {
                    fprintf(trace, "%s", "T43 ");
                    fprintf(stderr,
                         "ERROR--Range %d starts past end of reference
                         sequence\n", range_num);
                    return(0);
               }
               else
                    fprintf(trace, "%s", "F43 ");
          }
     }
     return(1);
}

...

main(argc, argv)
int argc;
char **argv;
{
     FILE *in, *out, *log, *genbank; /* files to be used */
     char *l;
     list orgs;                    /* list of organisms */
     organism taxa;                /* temporary organism pointer */
     organism reference;           /* reference organism */
     int reference_length;
     int alignment_length;   /* length of each alignment sequence */
     int new_length;         /* length of each alignment sequence */
     char *mask;             /* for storing which columns to use */
     int i;
     list previous_organism;
     int ch;
     int packing;
     int innum, genbanknum, outnum, toomany;
     int column_specified, sequence_specified, reference_specified;
     int column_error, sequence_error, reference_error;
     char *reference_name;
     int more;
     list seq_starts, seq_ends;
```

```
int col_start, col_end;
list seq_col_starts, seq_col_ends;
int error;

seq_starts=newlist();
seq_ends=newlist();
seq_col_starts=newlist();
seq_col_ends=newlist();

trace=fopen(tracefile, "w");

packing=1;
reference_specified=0;
reference_error=0;
innum=outnum=genbanknum=toomany=0;
more=1;
error=0;
for(i=1;i<argc && more;i++)
{
     if(argv[i][0]=='-')
     {
          fprintf(trace, "%s", "T0 ");
          if(compare(argv[i], "-help"))
          {
               fprintf(trace, "%s", "T1 ");
               printhelp(1, argv[0]);
               fclose(trace);
               exit(0);
          }
          else
          {
               fprintf(trace, "%s", "F1 ");
               if(compare(argv[i], "-pack"))
               {
                    fprintf(trace, "%s", "T2 ");
                    packing=1;
               }
               else
               {
                    fprintf(trace, "%s", "F2 ");
                    if(compare(argv[i], "-nopack"))
                    {
                         fprintf(trace, "%s", "T3 ");
                         packing=0;
                    }
                    else
                    {
                         fprintf(trace, "%s", "F3 ");
                         if(argv[i][1]=='c')
                         {
                              /* columns */
                              /* deal with later */
                         }
                         else if(argv[i][1]=='p')
                         {
                              /*position*/
                              /* deal with later */
```

```
                    }
                    else if(argv[i][1]=='r')
                    {
                         fprintf(trace, "%s", "T4 ");
                         /*reference*/

                         static int len, j;
                         static int error;

                         len=strlen(argv[i]);

                         for(j=0;j<len;j++)
                              if(argv[i][j]=='=')
                                   break;
                         if(j==len)
                         {
                              fprintf(trace, "%s", "T5 ");
                              fprintf(stderr,
                    "ERROR--No name specified on -r option\n");
                              printhelp(0, argv[0]);
                              fclose(trace);
                              exit(1);
                         }
                         else
                         {
                              fprintf(trace, "%s", "F5 ");
                              if(argv[i][j+1]=='\0')
                              {
                                   fprintf(trace, "%s", "T6 ");
                                   fprintf(stderr,
          "ERROR--Reference name on -r option is empty\n");
                                   printhelp(0, argv[0]);
                                   fclose(trace);
                                   exit(1);
                              }
                              else
                              {
                              fprintf(trace, "%s", "F6 ");
                    reference_name=dupstring(&argv[i][j+1]);
                              reference_specified=1;
                              }
                         }
                    }
                    else
                    {
                         fprintf(trace, "%s", "F4 ");
                         fprintf(stderr,
                    "ERROR--Invalid option: %s\n", argv[i]);
                         printhelp(0, argv[0]);
                         fclose(trace);
                         exit(1);
                    }
               }
          }
     }
}
else
```

```
        {
                fprintf(trace, "%s", "F0 ");
                if(innum==0)
                {
                        fprintf(trace, "%s", "T7 ");
                        innum=i;
                }
                else
                {
                        fprintf(trace, "%s", "F7 ");
                        if(genbanknum==0)
                        {
                                fprintf(trace, "%s", "T8 ");
                                genbanknum=i;
                        }
                        else
                        {
                                fprintf(trace, "%s", "F8 ");
                                if(outnum==0)
                                {
                                        fprintf(trace, "%s", "T9 ");
                                        outnum=i;
                                }
                                else
                                {
                                        fprintf(trace, "%s", "F9 ");
                                        toomany=1;
                                }
                        }
                }
        }
    } // end of for loop

    if(toomany || outnum==0)
    {
        fprintf(trace, "%s", "T10 ");
        if(toomany) {
                fprintf(trace, "%s", "T11 ");
                fprintf(stderr, "ERROR--Too Many file names\n");
        }
        else {
                fprintf(trace, "%s", "F11 ");
                fprintf(stderr, "ERROR--Not enough file names\n");
        }
        printhelp(0, argv[0]);
        fclose(trace);
        exit(1);
    }
    fprintf(trace, "%s", "F10 ");

    in=fopen(argv[innum], "r");
    genbank=fopen(argv[genbanknum], "r");
    out=fopen(argv[outnum], "w");
    log=stderr;
    if(in==NULL || out==NULL || log==NULL || genbank==NULL)
    {
        fprintf(trace, "%s", "T12 ");
```

```
        fprintf(stderr,
             "ERROR--Could not open files:");
        if(in!=NULL)
        {
             fprintf(trace, "%s", "T13 ");
             fclose(in);
        }
        else
        {
             fprintf(trace, "%s", "F13 ");
             fprintf(stderr, "\n\tinput ('%s')", argv[innum]);
        }
        if(out!=NULL)
        {
             fprintf(trace, "%s", "T14 ");
             fclose(out);
        }
        else
        {
             fprintf(trace, "%s", "F14 ");
             fprintf(stderr, "\n\toutput ('%s')", argv[outnum]);
        }
        if(log!=NULL)
        {
             if(log!=stderr)
                  fclose(log);
        }
        else
             fprintf(stderr, "\n\tlog ('%s')");
        if(genbank!=NULL)
        {
             fprintf(trace, "%s", "T15 ");
             fclose(genbank);
        }
        else
        {
             fprintf(trace, "%s", "F15 ");
             fprintf(stderr, "\n\talignment ('%s')",
                                       argv[genbanknum]);
        }
        fprintf(stderr, ".\n");
        fclose(trace);
        exit(1);
   }
   fprintf(trace, "%s", "F12 ");

   /* in file ok, genbank file ok, so start reading names */
   orgs=getnames(in, log);

   /* names read, so start reading their sequence entries */
   getentries(orgs, genbank, log);

   /* print error messages for names not found in genbank */
   startlist(orgs);
   taxa=listnext(orgs);
   while(taxa!=NULL)
    if(taxa->genbank_lines==NULL)
```

```
{
     fprintf(trace, "%s", "T16 ");
     /* name not in alignment, so remove it */
          if(taxa==reference)
               reference=NULL;
          fprintf(log,NAME_NOT_IN_GENBANK,taxa->name);
          free(taxa->name);
          if(listlastp(orgs))
          {
               rmcurr(orgs);
               taxa=NULL;
          }
          else
          {
               rmcurr(orgs);
               taxa=listcurr(orgs);
          }
}
else
{
     fprintf(trace, "%s", "F16 ");
     taxa=listnext(orgs);
}

startlist(orgs);
if(listlastp(orgs))
{
   fprintf(trace, "%s", "T17 ");
    /* no names found in alignment */
    fprintf(log,NO_NAMES_IN_GENBANK);
    fclose(in);
    fclose(out);
    if(log!=stderr)
     fclose(log);
    fclose(genbank);
    fclose(trace);
    exit(0);
}
fprintf(trace, "%s", "F17 ");

/* get reference organism with the range on it */
{
     list t;
     organism ecoli=NULL;    /*Susan : = added NULL*/
     organism first=NULL;
     char *s;

     reference=NULL;

     lfor(orgs, t)
     {
          if(first==NULL)
               first=thisorg(t);
          if(strcmp(thisorg(t)->name, "E.coli")==0) {
               fprintf(trace, "%s", "T18 ");
               ecoli=thisorg(t);
          }
```

```
            else
                fprintf(trace, "%s", "F18 ");
            if(reference_specified)
            {
                fprintf(trace, "%s", "T19 ");
                if(strcmp(thisorg(t)->name, reference_name)==0)
                {
                    fprintf(trace, "%s", "T20 ");
                    reference=thisorg(t);
                    break;
                }
                else
                    fprintf(trace, "%s", "F20 ");
            }
            else
                fprintf(trace, "%s", "F19 ");
        }

        if(reference==NULL)
            if(ecoli==NULL) {
                fprintf(trace, "%s", "T21 ");
                reference=first;
            }
            else {
                fprintf(trace, "%s", "F21 ");
                reference=ecoli;
            }
        reference_length=0;
        for(s=reference->sequence;*s!='\0';s++)
            if(base(*s))
                reference_length++;
    }

    /* get alignment_length */
    alignment_length=get_maximum_length(orgs);

    /* read column, sequence specification */
    sequence_specified=column_specified=0;
    sequence_error=column_error=0;
    more=1;
    for(i=1;i<argc && more;i++)
    {
        if(argv[i][0]=='-')
        {
            if(compare(argv[i], "-help"))
                /* dealt with before */;
            else if(compare(argv[i], "-pack"))
                /* dealt with before */;
            else if(compare(argv[i], "-nopack"))
                /* dealt with before */;
            else if(argv[i][1]=='c')
            {
                fprintf(trace, "%s", "T22 ");
                /* columns */

                static int len, j;
                static int error;
```

```
        if(column_specified)
        {
             fprintf(trace, "%s", "T23 ");
             fprintf(stderr,
             "ERROR--Columns specifed twice\n");
             printhelp(0, argv[0]);
             fclose(trace);
             exit(1);
        }
        fprintf(trace, "%s", "F23 ");
        len=strlen(argv[i]);
        for(j=0;j<len;j++)
             if(argv[i][j]=='=')
                  break;
        if(j==len)
        {
             fprintf(trace, "%s", "T24 ");
             fprintf(stderr,
             "ERROR--No columns specified on -c option\n");
             printhelp(0, argv[0]);
             fclose(trace);
             exit(1);
        }
        fprintf(trace, "%s", "F24 ");
        if(argv[i][j+1]=='\0')
        {
             fprintf(trace, "%s", "T25 ");
             fprintf(stderr,
             "ERROR--No columns specified on -c option\n");
             printhelp(0, argv[0]);
             fclose(trace);
             exit(1);
        }
        fprintf(trace, "%s", "F25 ");
        error=read_pair(&argv[i][j+1], &col_start, &col_end,
             1, alignment_length);
        if(error)
        {
             fprintf(trace, "%s", "T26 ");
             fprintf(stderr,
             "ERROR--Error reading column specification\n");
             printhelp(0, argv[0]);
             fclose(trace);
             exit(1);
        }
        fprintf(trace, "%s", "F26 ");
        column_specified=1;
}
else
{
        fprintf(trace, "%s", "F22 ");
             if(argv[i][1]=='p')
        {
             fprintf(trace, "%s", "T27 ");
             /*position*/
```

```
            static int len, j;
            static int error;

            if(sequence_specified)
            {
                 fprintf(trace, "%s", "T28 ");
                 fprintf(stderr,
            "ERROR--Sequence positions specifed twice\n");
                 printhelp(0, argv[0]);
                 fclose(trace);
                 exit(1);
            }
            fprintf(trace, "%s", "F28 ");

            len=strlen(argv[i]);
            for(j=0;j<len;j++)
                 if(argv[i][j]=='=')
                      break;
            if(j==len)
            {
                 fprintf(trace, "%s", "T29 ");
                 fprintf(stderr,
                 "ERROR--No list specified on -p option\n");
                 printhelp(0, argv[0]);
                 fclose(trace);
                 exit(1);
            }
            fprintf(trace, "%s", "F29 ");

            if(argv[i][j+1]=='\0') {
                 fprintf(trace, "%s", "T30 ");
                 fprintf(stderr,
                 "ERROR--No list specified on -p option\n");
                 printhelp(0, argv[0]);
                 fclose(trace);
                 exit(1);
            }
            fprintf(trace, "%s", "F30 ");

error=read_pair_list(&argv[i][j+1], seq_starts, seq_ends,
            1, reference_length, "Sequence Position list");
            if(error) {
                 fprintf(trace, "%s", "T31 ");
                 printhelp(0, argv[0]);
                 fclose(trace);
                 exit(1);
            }
            fprintf(trace, "%s", "F31 ");
            sequence_specified=1;
       }
       else
       {
            fprintf(trace, "%s", "F27 ");
            if(argv[i][1]=='r')
            {
                 fprintf(trace, "%s", "T32 ");
                 /*reference*/
```

```
                    /* dealt with before */
                }
                else
                {   /* Susan: reachable?, dealt with before */
                    fprintf(trace, "%s", "F32 ");
                    fprintf(stderr,
                    "ERROR--Invalid option: %s\n", argv[i]);
                    printhelp(0, argv[0]);
                    fclose(trace);
                    exit(1);
                }
                fprintf(trace, "%s", "F32 ");
            }
        }
    }
    else
    {
        /* files: dealt with before */
    }
}

if(!reconfigure_alignment(seq_starts, seq_ends,
    seq_col_starts, seq_col_ends,
    col_start, col_end,
    sequence_specified, column_specified,
    orgs, reference, &alignment_length, &mask))
{
    fprintf(trace, "%s", "T33 ");
    fclose(in);
    fclose(out);
    if(log!=stderr)
        fclose(log);
    fclose(genbank);
    fclose(trace);
    exit(0);
}
fprintf(trace, "%s", "F33 ");

if(packing)
{
    fprintf(trace, "%s", "T34 ");
    list start, end;

    /* remove common gaps, by filling mask with zeros */

for(start=firstnode(seq_col_starts),end=firstnode(seq_col_ends);
            start!=NULL;
            start=nextnode(start),end=nextnode(end))
        removegaps(orgs, mask,
            thisint(start), thisint(end),
            alignment_length);
}
fprintf(trace, "%s", "F34 ");

calc_num_base_pairs(orgs, mask, alignment_length);

startlist(orgs);
```

```
    while((taxa=listnext(orgs))!=NULL)
         printentry(taxa, mask, alignment_length, out);

    /* all done, so close files and quit */
    fclose(in);
    fclose(out);
    if(log!=stderr)
     fclose(log);
    fclose(genbank);
    fclose(trace);
    exit(0);
}
```