Evolutionary Divide and Conquer for the Set-Covering Problem

Luis Francisco González Hernández

Master of Science in Information Technology: Knowledge Based Systems
Department of Artificial Intelligence
University of Edinburgh

1995

Abstract

Genetic Algorithms provide a weak search method that scales rather badly when used in their traditional form. On the other hand, the set covering problem, when used to solve real-world problems, presents an enormous search space for genetic algorithms. A method is presented, by which genetic material is maintained for a long time within the population to allow a greater portion of the search space to be covered in the hope of producing greater quality solutions. The problem is broken down into smaller subproblems for which a population of solutions is evolved. The quality of those solutions is then assessed depending on how well they fit in with solutions to other subproblems. We find that the method has merit and potential, and we discuss its sensitivity to various parameters.

Acknowledgements

First of all, I would like to thank my supervisor, Dave Corne, for his availability, his friendliness, his support, and his calmness in the most desperate moments, and for his full confidence in me. He has always been a source of reassurance and encouragement. I would also like to thank Luisa, for showing infinite patience and the will to always listen to me in the most troublesome moments of this year—who can ask anybody for more? Further I want to express my gratitude to my parents, my brother, José Miguel, for his proofreading, and especially my sister Carmen. They alone understand how much help and support they have shown to me, and how important my "weekend escapes" were to succeeding in this work.

I would also like to thank Paul C. H. Chu, who kindly agreed to provide me with the results of the problems of the *OR-Library*, reported in [Beasley & Chu 95]. My gratitude also to the *Student Awards Agency for Scotland*, which partially supported my studies here at the University of Edinburgh through the grant D7575254/D32. Thanks also to John, who bore with my insistent questions on the correct usage of the English Language. With out your help, this would certainly be a nightmare to read.

Last (but not least) I would like to thank Bety, Dae Gyu, Laura, Margarita, Olga, Roman, and especially Carlos, Delmi and Alejandro—you sure know what friends are. Rest assured I will never forget you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Evolutionary Divide and Conquer

1.1 Concept: Retaining Variety

The success of a genetic algorithm in finding at least a near optimal solution depends on the amount of genetic material the population is capable of maintaining throughout the run, as explained below in section 2.3. The algorithm mainly re-shuffles the information contained in the population, relying on mutation to produce new material that is not originally in it.

There are several ways proposed in the literature to accomplish this end. An obvious one just makes the population of solutions bigger so that there are more opportunities to have varied material within the population. This essentially means that we reduce the genetic pressure, because we are allowing less fit individuals to remain in our pool.

The problem with this approach is that at the end of the day we are just gaining some time for those individuals, but as the algorithm progresses, they are bound to disappear, and the whole population will eventually converge to some possibly local optimum.

Nature seems to allow several fit species to coexist, each in its own way a fit population and nevertheless completely different from the others, yet showing a great deal of adaptation. The process of generating two species from a common ancestor population is called *Speciation*. Another possible solution in order to retain variety, and one that allows the genetic algorithm find several maxima, is forcing niches[1]. Several solutions have been suggested in the literature. [DeJong 75] proposed a population replacement strategy called *crowding*, which in essence forces death upon that individual that most closely resembles the new offspring that is being introduced in the population. To accomplish this a small sample of individuals is compared with the new offspring. In the case of binary-coded chromosomes the measure of "similarity" used, was the *hamming distance*[2] although the particular criterion used depends on the coding.

Another attempt at a solution described in [Whitley & Starkweather 90] consists in keeping several isolated populations that are initialised randomly. Only a small amount of transference of genetic material is allowed by permitting some individuals to "migrate" from one population to another at regular intervals. This at least facilitates, if it does not guarantee that the genetic algorithm can converge to several different solutions, one per population. The method, of course, lends itself ideally to a parallel-processor implementation.

This seems more efficient in terms of the quality of the solution than both using completely isolated populations or a big one whose total number of individuals is the same. This method is referred to as the *island model* (see [Whitley 94]).

Cellular genetic algorithms constitute another proposed scheme reported in [Davidor 91]. It consists in keeping a geographic distribution of individuals which are only allowed to mate locally. In this way diversity is maintained longer, because it is not possible for good material to spread so fast through the population.

[1] *niche* in this context refers to a population that speciates, possibly becoming completely different from the others. Here we draw the analogy to the corresponding natural mechanism.

[2] For binary-coded chromosomes, the hamming distance is the number of genes that have different alleles.

Nonetheless, the model does not prevent the population from converging in the end.

The present work proposes and investigates a new way of dealing with this problem. Instead of having a population of solutions, the problem to be solved is broken into several smaller subproblems, and each of those is assigned a pool of solutions. So each chromosome is just a partial solution to the original problem.

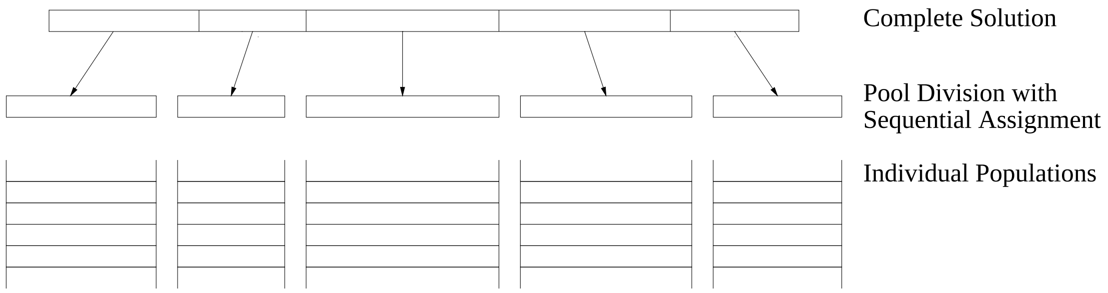

Figure 1–1: Divide and Conquer

The process is shown in Figure 1–1. Suppose we have a way of coding a solution in a chromosome in the way that would be used for a traditional genetic algorithm. According to some external rule, possibly just at random, we divide the problem into several subproblems, i.e., the coding of a solution is divided so that certain parts of it represent a subproblem we would like to solve. These new subproblems are allocated a population that we will evolve, keeping the material for that problem isolated from the rest.

In order to produce a valid solution to the overall problem, individuals from each population are chosen according to some specific rules and combined to form an overall solution. This in turn is evaluated and a fitness score is then assigned to each element, which is otherwise evolved independently within its own population.

1.1.1 Virtual Population

The effect of the proposed representation is that potentially there is a big virtual population used to search the solution space. The rationale is as follows. To build a complete description of a problem's solution, one chromosome of each pool[3] is selected in turn, and combined with the others. This means that the total number of possible combinations, assuming that all individuals within the population are distinct, is

$$Virtual\_Population\_Size = Pool\_Size^{Number\_of\_Pools} \tag{1.1}$$

Pool_Size	Number_of_Pools	Virtual_Population
25	5	9765625
25	15	9.313225e+20
50	5	3.125e+08
50	15	3.051757e+25
50	25	2.980232e+42

Table 1–1: Virtual Population Sizes

As an illustration, Table 1–1 shows some *Virtual_Population_Sizes* for different values in *Pool_Size* and *Number_of_Pools*. Of course, the majority of those potential members of the population are never evaluated, or else the genetic algorithm would take too great an amount of time to complete. But ensuring that each element gets a score that is in some way representative of how well it would score with the individuals in other pools should greatly enhance the performance of the genetic

[3] We will refer from now on to the populations trying to solve a subproblem as a solution pool.

algorithm. As we will see, the basic model turns out not so effective as it may seem, but this has mostly to do with the tradeoffs that have to be met in order to produce a feasible algorithm.

1.2 Overview

This dissertation is organised as follows:

Chapter2, *Background*, introduces the basics of genetic algorithms, describing various designs that have been proposed in the literature. The set covering problem is also formally defined, and some information on several other solution algorithms is given.

Chapter 3, *Implementing DCSCPGA* describes the developed system in more detail. Several design options are described along with their trade-offs, and the essential decisions are explained.

Chapter 4, *Experiments*, introduces the objectives set up. The test problems are described in detail and the specific test plan is exposed.

Chapter 5, *Preliminary Experiments*, describes the first results obtained, and reviews in great detail how the different parameters influence the genetic material found in the population. The main goal was to decouple the performance of the approach from the influence of other parameters.These results were used to revise the program in the hope of improving the performance.

Chapter 6, *Results*, discusses the actual performance of the algorithm on the test problems and the influence of the different parameters.

Chapter 7, *Conclusions*, summarises the findings and critically evaluates the performance observed, and several possible lines to extend the work are presented.

Chapter 2

Background

2.1 Genetic Algorithms

2.1.1 Concept

Genetic algorithms first proposed by Holland [Holland 75], are weak stochastic computational search techniques based on the theory of evolution as stated by *Charles Darwin* in his theory of natural selection.

> "Genetic algorithms use a direct analogy of natural behaviour. They work with a population of "individuals", each representing a possible solution to a given problem. Each individual is assigned a "fitness score" according to how good a solution to the problem it is. (...) The highly fit individuals are given opportunities to "reproduce", by "cross breeding" with other individuals in the population. This produces new individuals as "offspring", which share some features taken from each "parent". The least fit of the members of the population are less likely to get selected for reproduction and "die out"."

The above quotation taken from [Beasley *et al.* 93] underlines the key concepts involved in genetic algorithms. First of all the search is carried out maintaining a population of possible solutions, each of which is allocated a chance to breed according to a measure of how good a solution it is to the problem. Each genetic algorithm embodies a procedure that can be stated as in Figure 2–1

Procedure: *Genetic Algorithm*

```
Initialise:

    t ← 0
    initialise Population(t)
    evaluate Population(t)


while(not(termination_condition))
{

    t ← t + 1
    select parent1 and parent2 from Population P(t)
    offspring ← cross parent1 and parent2 with
                probability crossover_rate
    mutate offspring with probability mutation_rate
    evaluate offspring
    insert offspring in Population(t)


}
```

Figure 2–1: Pseudo-code for a Genetic Algorithm

2.1.2 Characteristics of Problems appropriate for Genetic Algorithms

In order for a problem to be appropriate for a genetic algorithm, it has to have the following two properties:

- First of all, the problem should lend itself to coding as a string of parameters, either using the binary coding suggested originally by [Holland 75] and others, or using a non-binary alphabet, as we suggest here following [Michalewicz 94]. The population of solutions are actually collections of those strings that are recombined and modified according to Figure 2–1. Each parameter or (in the case of binary codings) each bit is called a *gene*. The values that a gene can have are called *alleles*, and correspond to the possible values of the parameters. The collection of all the genes constitute the *genotype*, and the solution reconstructed from the coding is the *phenotype*, using again the analogy to natural genetics.

- There is also the need for a function that quantifies the idea of quality of a solution. As said above, the reproductive chances are allocated according to the fitness of the solution, which in turn is determined using that function.

2.1.3 Building Block Hypothesis

The most well known account so far of why genetic algorithms work is based on the *Building Block Hypothesis*, originally devised by Goldberg and summarized in [Michalewicz 94], which states that the genetic algorithm seeks "a near optimal performance through the juxtaposition of short, low-order, high performance schemata" (building blocks).

The recombination process consists in mixing the genotypes of the parents in the hope that recombining good solutions will produce an offspring that inherits the good qualities of each parent, and itself becomes an even better solution.

Sometimes a small change in the genotype is introduced through mutation. This process is based on the assumption that small blocks within the genotype determine the fitness of the solutions. Those blocks, called schemas, are processed by the genetic algorithm in a highly parallel fashion.

A schema represents all genotypes that contain it, and is characterized by the presence of certain genes in some specific order. The values of other genes are irrelevant.

Schema: 1 0 * * 1 * * * 0 *

Figure 2–2: Representation of a Schema

For example, Figure 2–2 represents a schema which generalizes all the possible codings that start with `1 0`, have a `1` in the fifth gene, and a `0` in the ninth position, a total of $2^5 = 32$ chromosomes for a binary coding.

There are two important magnitudes defined in relationship with a schema:

Defining Length The defining length of a schema is the distance between the first and the last fixed gene.

Order The order of a schema is defined as the number of fixed genes.

The greater the order of the schema, the more specific it is. On the other hand, the greater the defining length, the greater the probability that it is disrupted during the crossover operations. It gives the compactness of the information contained in the schema.

There is abundant empirical evidence that the *building block hypothesis* is true for most real-world optimisation problems. This of course does not mean that it always works. There is an important type of problems called *deceptive problems*, for which a genetic algorithm is incapable of finding near-optimal solutions. Their most important characteristic is that short promising building blocks are actually misguiding the search and leading the genetic algorithm away from the optimal solutions. [Goldberg 89] suggests that deceptive problems seem to involve codings and fitness functions that contain isolated optima surrounded by the worse solutions.

The fitness landscape in Figure 2–3 is an example of such a deceptive problem. There is no information in the fitness function that evidences a maximum near to the optimum. Except in the highly unlikely event of having that maximum in the initial population, the genetic algorithm has no information that leads it to that optimum.

2.1.4 Selection

As sketched in Figure 2–1, the evolutionary process starts by selecting the solutions that are going to be recombined. The selection process stochastically samples the population and determines the direction in the evolution process. Several different selection schemes have been implemented and are reported in the literature. The most important are:

Fitness Proportional Selection This selection scheme allocates the reproduction chances proportional to the ratio between the fitness of the individual to the average fitness of the population. This scheme is normally called roulette wheel selection.

The problem with this method is that it is affected by the range of the fitness values in the population. The greater those are, the smaller the relative fitness difference, which in turn means more equal selection opportunities.

Rank based selection This scheme tries to circumvent the problem found in fitness based selection by ordering the individuals and assigning the reproductive opportunities according to some function of the rank.

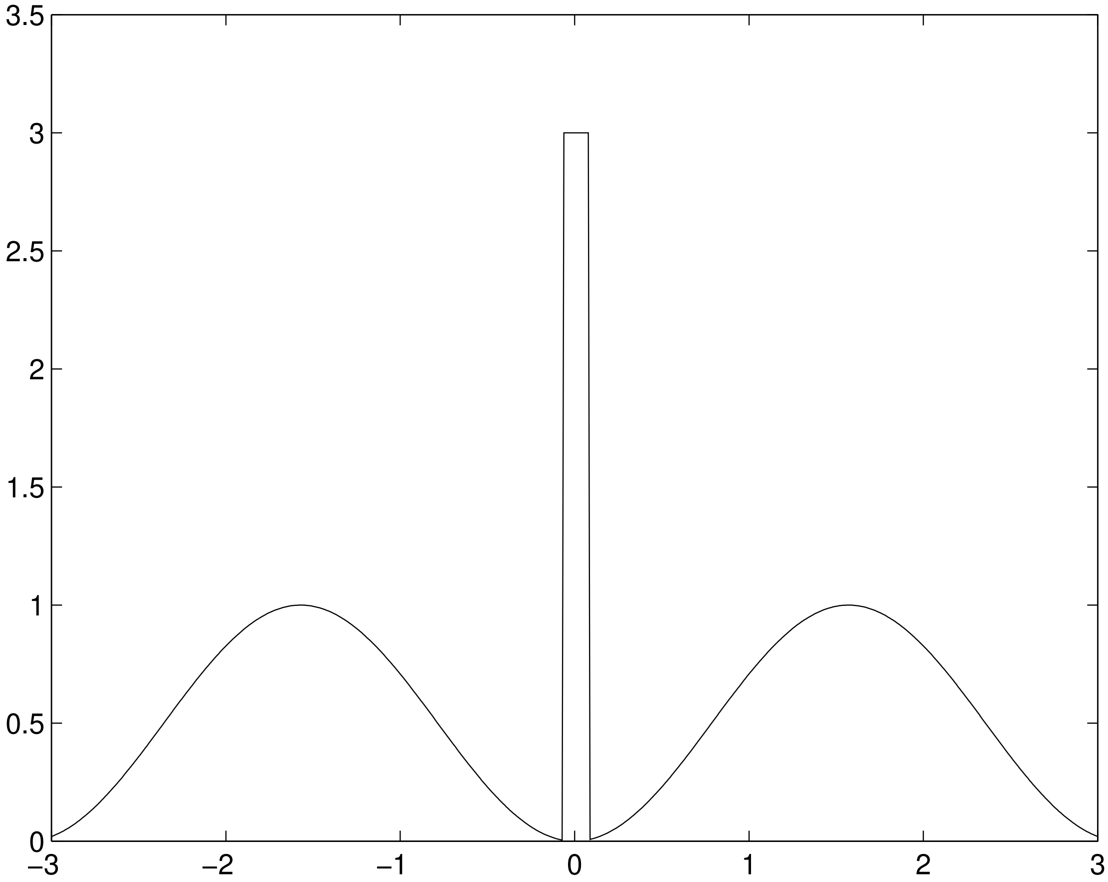

Figure 2–3: Deceptive Landscape

The scheme implemented in the present work[1] selected the element using the root of a parabolic function of the rank, a parameter fixed for every run, and a random number, using the following equation:

$$rank = int\left(Size\_of\_Population\frac{bias - \sqrt{bias^2 - 4(bias-1)rand}}{2(bias-1)}\right) \quad (2.1)$$

[1] This scheme is original of PGA © Peter Ross and Geoff Ballinger, 1993 originally based on GENITOR [Whitley 89].

where "bias" is a parameter determined at the beginning of the run and "rand" is a random number uniformly distributed in the interval [0,1).

Tournament Selection This scheme, suggested in [Brindle 81] along with several others, is intended to produce better results. Originally, the method would calculate the probabilities of selection in the usual way, as in fitness proportional selection, and successive pairs of individuals were selected, promoting for reproduction the fittest within the pair.

In this work tournament selection is implemented by selecting a group whose size is fixed for each run, and within the selected group by promoting the fittest as parent. There is also a slightly varied scheme that searches up to a certain number of individuals and tries to locate one that has greater fitness than the first one. Once one such chromosome is found, it is selected as parent. If on the other hand none is found after a predetermined number of times, set for each run, the first picked chromosome is selected as parent.

2.1.5 Genetic Operators

A genetic algorithm produces the new solution using *genetic operators* to combine the building blocks of fit *parents*, in the hope that the newly created offspring will inherit the building blocks that made the ancestors fit, and thus become better solutions. Multiple operators have been proposed in the literature, the most important ones being crossover and mutation.

2.1.5.1 Crossover

Crossover mixes the genetic material found in the parents, in the hope that having highly fit ancestors will produce potentially better offspring. This fact is expected to lead the search through promising regions in the solution space. The idea is that deriving the *genotype* as combination of fit individuals will combine their good characteristics together in the new offspring.

Several different crossover schemes have been proposed, the most important being:

One-point crossover As shown in Figure 2–4, a point in the genotype string of the parents is chosen at random, and the offspring's genotype will be the one of the first parent up to that point, and that of the other after it.

Parent 1	A	B	C	D	E	F	G	H	I	J
Parent 2	a	b	c	d	e	f	g	h	i	j
Offspring	A_1	B_1	C_1	D_1	E_1	F_1	g_2	h_2	i_2	j_2

Figure 2–4: One-point Crossover

Two-point crossover In this case two points are chosen within the genotype string of the parents. Figure 2–5 shows how the offspring inherits the genetic material between those points from the second parent, and the rest from the first.

Parent 1	A	B	C	D	E	F	G	H	I	J
Parent 2	a	b	c	d	e	f	g	h	i	j
Offspring	A_1	B_1	C_1	d_2	e_2	f_2	G_1	H_1	I_1	J_1

Figure 2–5: Two-point Crossover

Multiple-point crossover Generalizing the scheme presented in the previous crossover operators, now several points are chosen that allow several crossover points to be used.

Uniform crossover As shown in figure 2–7, each gene in the offspring has the same odds to be inherited from any of both parents. To produce the new

Parent 1	A	B	C	D	...	F	G	H	I	J
Parent 2	a	b	c	d	...	f	g	h	i	j
Offspring	A_1	b_2	c_2	D_1	...	F_1	g_2	h_2	i_2	J_1

Figure 2–6: Multiple-point Crossover

offspring a random mask is created, which determines which parent is chosen for each gene.

Parent 1	A	B	C	D	E	F	G	H	I	J
Parent 2	a	b	c	d	e	f	g	h	i	j
Random Mask	1	1	2	2	1	2	1	1	2	1
Offspring	A_1	B_1	c_2	d_2	E_1	f_2	G_1	H_1	i_2	J_1

Figure 2–7: Uniform Crossover

There is no conclusive evidence as to which crossover operator is better. [Syswerda 89] argues that whereas two-point crossover is more likely to break up larger building blocks, uniform crossover treats all in almost the same way, irrespective of their defining length. At the same time uniform crossover makes unnecessary the use of reordering operators like inversion[2].

[2]The order of the genes in the chromosome has a great impact on the crossover efficiency. Schemas are disrupted in proportion to their order. So, several reordering mechanisms have been developed to allow the genetic algorithm to search for the best order in the chromosome string. Inversion (see [Goldberg 89]) acts by selecting two points in the string and reversing the order in those genes. This would change the string

Crossover operators like one or two-point crossover are dependent on the ordering of the genes, which does not necessarily have to be the best possible. Uniform crossover handles all genes in any possible location in the same way. It seems therefore that uniform crossover is more suited to problems having greater chromosome lengths.

A more extensive study reported in [Eshelman *et al.* 89] suggests that there is no such thing as a *preferable* crossover operator. One, two, multiple, and uniform crossover were analysed both theoretically and empirically using several problems, yielding no clearly advantageous scheme. All operators presented a similar performance on the test problems.

2.1.5.2 Mutation

Mutation is another important operator that randomly changes the alleles present in the genotype. The expected number of changes, called *mutation rate*, is either fixed or varies according to a certain criterion. The latter is referred to as *adaptive mutation*.

Traditionally researchers have considered mutation as a *background* operator [DeJong 85]. Its importance lies in its ability to allow for a certain amount of random search, so that important genetic material that is either not present at all in the initial population, or has been discarded during earlier phases of the search, can be reintroduced into the current population. This allows the genetic algorithm to explore new regions in the search space, and enables it to overcome local minima.

There is some debate about which operator, mutation or crossover, is more important. Nature suggests that mutation is capable of producing complex or-

---

{1 2 * 3 4 5 * 6} into {1 2 * 5 4 3 * 6 }, thus altering the order and the defining length of the schemas.

ganisms by itself, and biologists consider it as the main drive in the evolutionary process. There is evidence that suggests that although mutation-only evolution is slower than combining it with crossover, in general it produces better quality solutions than a crossover-only algorithm [Schaffer & Eshelman 91].

In traditional genetic algorithms that use a binary coding, mutation would just mean flipping a certain bit. The use of non-binary alphabets forces the extension of this concept. Here, a random number up to the number of possible alleles for a gene would be generated, and the corresponding allele would be selected according to a given order.

There is a conflict between the mutation rate and convergence. The smaller the mutation rate, the faster convergence is attained. This is at the cost of being more vulnerable to getting stuck in local minima. On the other hand, at greater rates convergence may not happen. Adaptive mutation schemes, although not sufficiently developed so far, seem to provide the right sort of framework to reach a compromise. As in *simulated annealing* mutation could decrease the expected number of changes with time, allowing the algorithm to settle down and eventually converge. Sometimes, though, it is more interesting to increase mutation when convergence is found to force the algorithm out of a local optimum.

2.2 The Set Covering Problem

The set covering problem can be formulated as covering at a minimal cost all the rows of an $m \times n$ matrix of ones and zeros by a subset of columns. If we consider the vector $\vec{x}$, such that $x_i = 1 \Leftrightarrow \textit{column } i$ is in the solution, and $x_i = 0$ otherwise, we can then formulate the problem as in Figure 2–8.

Many applications of the set covering problem to the modelling of real-world problems have been described. Most often it has been used for optimisation, especially for resource allocation [Garey & Johnson 79] and scheduling

Minimise:

$$\sum_{j=1}^{n} c_j x_j \tag{2.2}$$

Subject to:

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1, \forall i \in \{1, ..., m\} \tag{2.3}$$

$$x_j, a_{ij} \in \{0, 1\}, \forall i \in \{1, ..., m\}, \forall j \in \{1, ..., n\} \tag{2.4}$$

Figure 2–8: The Set Covering Problem

[Marsten & Shepardson 81]. In particular, several problems used in the present work are bus-driver scheduling problems. In these the rows represent the services that must be offered, and the columns or sets are the driver shifts that might cover them. The matrix elements mark those services that can be covered by each shift.

$$a_{ij} = \begin{cases} 1 & \text{if driver shift i can cover service j} \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

There are a number of both optimal and heuristic solution algorithms that have been reported in the operations research and mathematical literature during the last years, including an optimal solution method developed for small problems by Fisher and Kedia [Fisher & Kedia 90] based on a dual heuristic. Other solution methods discussed in [Balas & Ho 80] include integer programming and heuristic branch and bound. As with other sizeable management problems, there has recently been an increasing interest in evolutionary computation solution techniques. [Jacobs & Brusco 93] report considerable success in an heuristic solution for big problems based on simulated annealing, and [Beasley & Chu 95] propose a genetic algorithm based approach. Earlier attempts using genetic algorithms include [Liepins & Hilliard 87] and [Liepins & Potter 91]. For a detailed account of the application of evolutionary techniques to management problems, including the set covering problem, see [Nissen 93].

2.3 Practical Genetic Algorithms: Departing from Ideal

Genetic algorithms combine the use of knowledge about the search space already visited with a certain amount of random search in a way that proves to be optimal under certain assumptions [Holland 75].

Firstly, it is assumed that an infinite population is kept. For obvious reasons this cannot be accomplished in practice. It is further assumed that the fitness function provides an accurate estimate of the usefulness of a solution, and that there is negligible interaction between the genes within a chromosome.

2.3.1 Genetic Drift

The first assumption produces several departures from the ideal in the behaviour exhibited by the genetic algorithms. Most selection schemes (among them several discussed earlier) try to minimise the stochastic sampling error due to the finiteness of the population.

But this is not the only problem. Imposing a limit on the size of the population limits the amount of different alleles found in the population. There is a *selection pressure* to promote fitter individuals which are more likely to be present in the following generations. But some of the material needed may be within less fit individuals that are selected to die out. If a certain allele is needed but is only in low-fitness individuals, it may disappear from the population, preventing it from reaching optimal solutions. It is then up to mutation to regenerate that material by pure random search. To illustrate this argument consider the *Max Problem*. The task is to find a binary coded string that has the maximum possible number of ones. Fitness is measured in terms of the number of ones present in the string. Consider the 16-bit chromosome population sketched in Figure 2–9:

Number of Chromo	Genotype	Fitness
Chromo 1:	[1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 1]	13
Chromo 2:	[1 1 1 0 1 1 0 1 0 1 1 1 1 1 1 0]	12
Chromo 3:	[1 1 1 0 1 0 0 1 0 1 1 1 1 1 1 1]	12
...	...	...
Chromo i:	[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]	1
...	...	...

Figure 2–9: Genetic Drift in the context of the Max Problem

We can see that chromosome i is very likely to be selected to die out, whereas the first three are fitter and probably will go into the next generation, spreading their genetic material. But only chromosome i holds the information relevant to increase the fitness of those first three. Once this individual dies, only a mutation of the fourth gene into a one will further enhance the solution. This means that in this case the pressure to select the fittest preventing the algorithm from keeping valuable genetic material. This phenomenon, also observable in nature is called *Genetic Drift*. It would be avoided with an infinite population, as the chromosome in this case would not have to die.

When a local maximum is reached or when, as in the max problem, the absence of varied genetic material in the population makes it impossible to progress any further, the population is said to *converge prematurely*.

2.3.2 Implications of the Population Size

The size of the population modifies the effect of genetic drift: the greater the size of the population, the greater the amount of genetic material it can keep. Therefore, it is important that the population be big enough to prevent the genetic algorithm from converging too fast and producing suboptimal solutions. On the other hand, if the population is too big evaluation will take too long. This means that there has to be a trade-off between the amount of genetic material in the population and the time it takes the genetic algorithm to run.

Specifically, when dealing with a set covering problem the size of the population should be big enough to allow the genetic algorithm enough information to search the whole solution space evenly. Nevertheless, this is impossible for bigger problems—those involving a big number of rows and columns. Thus, the initial population should be selected using a heuristic that aims to produce enough relevant individuals. Following [Beasley & Chu 95], given a set covering problem with density[3] Φ, a solution will have on average $1/\Phi$ elements, and so the initial population will have *Popsize* $\times 1/\Phi$ columns. To cover the population and to ensure a reasonable variety, this number should be at least greater than the number of columns in the matrix, preferably several times that number. Although real-world set covering problems have rather sparse matrices, a density of 5% with 2000 columns would demand a population of at least several times $0.05 \times 2000 = 100$. This means that the populations have to be very big, or a special mechanism has to be implemented for a genetic algorithm to be of any use. Specifically, [Beasley & Chu 95] propose the use of an heuristic in the hope that the initial population is drawn from a reduced number, but hopefully a useful subset, of the possible solution strings.

[3] *density* is the ratio of active elements to the total in the matrix that defines the problem.

Chapter 3

Implementing DCSCPGA

3.1 Introduction

In order to test the divide and conquer approach, the set covering problem as described in section 2.2 was selected. This provides a real-world application that presents a combinatorial search space, and thus seems to be the kind of problem our method tries to address. The constructed system, called DCSCPGA for *Divide and Conquer Set Covering Problem Genetic Algorithm*, is based on a modification due to Dave Corne of PGA, the Parallel Genetic Algorithm originally from Peter Ross and Geoff Ballinger, used to solve the set covering problem. The modifications are covered by section two of the GNU public license.

3.2 Coding the Set Covering Problem

3.2.1 Binary Coded Chromosomes

The first thing to decide when implementing a genetic algorithm is the way the solution parameters are going to be coded in a chromosome. The implementation

1	2	3	4	5	...	n-2	n-1	n	Column - Gene
1	1	0	0	1	...	0	1	1	Chromosome

Figure 3–1: Binary representation of the Set Covering Problem

used in [Beasley & Chu 95] uses the coding shown in Figure 3–1, which uses one gene per column or set in the cover. The genes are binary-coded, and the alleles are one if the set is in the cover, and zero if not. This coding scheme is straightforward, but has the disadvantage that some chromosomes, in fact the majority, represent infeasible solutions. For an individual to represent a feasible solution to the problem, all rows should be covered by at least one set.

Rows	Set One	Set Two	Set Three	Set Four	Set Five
1	1	0	0	1	0
2	1	0	0	1	0
3	0	0	1	0	1
4	0	0	1	0	1
5	0	1	0	1	0
6	0	1	0	0	1
Cost	10	10	30	20	20

Table 3–1: Sample Set Covering Problem with six rows and five sets.

Parent One	0	0	0	1	1
Parent Two	1	1	1	0	0
Offspring	0_1	0_1	1_2	0_2	0_2

Figure 3–2: Reproduction using a binary representation

Given the problem specified in Table 3–1, and supposing we carry out a reproductive step using the one-point crossover as illustrated in Figure 3–2, where the crossover point has been set between the second and third genes, then the offspring produced is not a feasible solution. This new individual only covers the third and fourth rows. This means that the algorithm has to implement a way of circumventing this problem. There are several methods that have been used to accomplish this (see for example [David & Steenstrup 87]) all of which fall within one of three categories:

High Penalty In this case the fitness function assigns a high penalty to those individuals that represent non-feasible solutions. This makes them die out and disappear from the population. Such a method, though, can be unsatisfactory. It could happen that infeasible individuals are more likely to be produced, which would make the genetic algorithm waste most of the time evaluating useless chromosomes. It can even be the case that once a possible, though maybe unfit, solution is found, the population rapidly converges to it because of the difference in fitness compared to the non-feasible specimens. Or maybe there is just no way at all to produce an optimal solution without first having to go through infeasible individuals.

Moderate Penalty Here the choice is to have moderate punishments to avoid the difficulties outlined for high penalty schemes. The risk is that the value is not sufficient to avoid the population from prefering infeasible, yet well-rated individuals.

Decoding The third possibility is having an elaborate decoding, responsible for mapping the infeasible representations to allowed solutions. However, this method tends to be computationally expensive. The complexity avoided using a simple representation is just shifted to the decoding phase.

[Beasley & Chu 95] use an heuristic to guarantee that the produced individual will constitute a feasible solution. This is done by adding columns that take care of uncovered rows according to their heuristic value, based on the ratio of individual cost to number of uncovered rows it caters to.

3.2.2 Non-binary coding

1	2	3	4	5	...	n-2	n-1	n	Row - Gene
C_1	C_2	C_3	C_4	C_5	...	C_{n-2}	C_{n-1}	C_n	Chromosome

Figure 3–3: Non-binary Coding for the Set Covering Problem

As an alternative to having to re-process the offspring produced by the genetic algorithm, a non-binary coding could be used. To explain the idea we will refer again to the sample problem described in Table 3–1. In this case, as shown in Figure 3–3, each gene in the chromosome represents a row in the problem. The sets covering each row are ordered as the data of the problem is read, and the greatest value for an allele is set to be the number of coverers each row has. So the value of the allele itself represents the C_i-th set that covers the row of the chromosomes in Figure 3–4. This procedure always generates feasible offspring by crossover.

This option represents a trade-off. On the one hand, this coding guarantees that any offspring produced is a feasible solution, but on the other hand there is some amount of decoding necessary in order to evaluate the fitness of an individual. The problem is that a subset of the sets specified in the chromosome might already be a cover, and therefore a redundancy check must be carried out every time, making the process more complex. In the example presented in Figure 3–4 we see that the cover coded in the offspring is $\{1,2,3,4\}$, but the subset $\{1,2,3\}$ is already a solution. Therefore the cost of set 4 should not be added when computing the fitness of the individual.

Parent One	1	1	2	1	2	2
Parent Two	2	1	1	1	2	1
Offspring	1_1	1_1	1_2	1_2	2_2	1_2

This corresponds to the covers:

Rows	Parent One	Parent Two	Offspring
1	Set 1	Set 4	Set 1
2	Set 1	Set 1	Set 1
3	Set 5	Set 3	Set 3
4	Set 3	Set 3	Set 3
5	Set 4	Set 4	Set 4
6	Set 5	Set 2	Set 2

Figure 3–4: Reproduction using a non-binary representation

3.3 Evaluating the Fitness of a Chromosome

3.3.1 The Alternatives and their trade-offs

The divide and conquer approach requires the development of a new evaluation method. There is the need for a strategy that selects each chromosome based on a representative score of how well it performs when combined with the populations in other pools. Yet the evaluation function is critical in the performance of the genetic algorithm in terms of computational resources. A pure sequential evaluation, in which each chromosome is scored using all combinations, is obviously not possible in view of the number of combinations it involves (see Table 1–1). On the other hand, for the algorithm to progress the level of co-adaptation needs to be measured in some way. Just one score is not enough. In a certain respect this is not unlike the credit assignment problem. A very good chromosome might, when combined with a very bad one, average the score, allowing the bad one to survive although itself not fit enough. On the other hand, an average individual combined with a bad one might die out instead. This means that a compromise between the representativeness of the fitness score and the computational complexity of the evaluation process has to be met.

3.3.2 Generational versus Steady State Genetic Algorithms

The replacement strategy for a genetic algorithm not only decides which are the individuals that shall be substituted by the new offspring, but it also establishes in which way the transition from one population to another is carried out. Sometimes the selection process creates a so-called *mating pool*, which as explained earlier allocates greater opportunities to the fittest individuals. Once the mating pool is created, reproduction using crossover and mutation is carried out, producing as many offspring as individuals in the population. Then those new chromosomes become the next population that substitutes the original one. This replacement strategy characterizes the so-called *generational genetic algorithms*. On the other hand, the *steady state genetic algorithms* immediately insert a newly bred individual in the population.

Comparing both methods it seems that the dynamics of generational genetic algorithms allows the system to evolve but not necessarily improve more rapidly, whereas the steady state scheme keeps a more stable population. Theoretically, a generation approach might worsen the fitness of the whole population in one generation unless an *elitist* replacement scheme is implemented that prevents the fittest individuals from dying out.

3.3.2.1 Generational Evaluation Schemes

Due to the nature of the divide and conquer approach there may be some interest in the generational approach. As we said before, chromosome fitness in some way measures the degree of co-evolution of the pools. This means that a slight change in one pool can have its effect dramatically sharpened because of the genetic content of the population in other pools. So, after each offspring is generated, the fitness of the rest of the individuals is somewhat influenced, making their scores out-of-date. Therefore they have to be recalculated, so that they may faithfully resemble the actual score of the individual in the pool. Of course, one way of doing this would be to create a new complete solution population and to replace the old one, reevaluating each chromosome. The hope is that the computational cost is not too high and that the genetic algorithm converges faster, having the divide and conquer approach guarantee the presence of enough genetic material to provide optimal solutions.

This method was implemented, but showed, already after little experimenting, to be computationally inefficient when compared to the scheme finally chosen as optimal.

3.3.2.2 Individual Evaluation Schemes

As opposed to the generational one, steady state evaluation schemes could be implemented requiring that the populations' fitness be assessed every generation. This is clearly inefficient. On top of that, there is the need to assign a score to the new offspring. To calculate the fitness of a chromosome, it will be repeatedly combined with others, and the average score will be assigned as its fitness.

3.3.3 Implementing the compromise

To avoid the overhead of having to recompute the fitness of every member of the solution population, a compromise was reached, which required the population to update the scores every five generations. The actual number was set in view of the population size used in the experiments, 50 chromosomes, which makes five amount to 10%. Several trials showed that there was no appreciable difference in the convergence between genetic algorithms with exhaustive update and those using the compromised 10% delay while the required number of evaluation steps was four times smaller.

3.3.4 Isolated and Combined Evaluation

Although not implemented in the end, another scheme was also considered. The idea is to have interleaved periods, in which the subproblem pools are left to evolve independently and to try to solve their specific subproblems; and then to combine them in an attempt to smooth out the specificities that make highly specialized individuals unfit for global solution. Here we rely more heavily on the combined evolution but allow the population from time to time to get better at their problem in order to attain faster convergence. However, this seems to be useful only for the initial stage of the algorithm; so it would have to be used sparsely as more and more generations are evolved.

3.4 Assigning the fitness: the active gene

The coding scheme implemented is such that any row in the problem is assigned a cover. Genes are interpreted one by one, from left to right, each indicating a new set that needs to be added to the cover.

Chromosome	1	2	1	2	1	2
Cover	Set 1	Set 4	Set 3	Set 5	Set 4	Set 5

Figure 3–5: Explaining the concept of active gene.

However only *active* genes are interpreted. Initially all genes are considered active. During the interpretation process though some are rendered inactive. Suppose that the process reaches the i-th gene. The set indicated by the gene, s_i is added to the cover and any other gene j such that $j > i$ is deactivated. Once this is done, all genes still active are checked to ensure that there are no redundant sets in the cover. A redundant set may appear if there are various sets that cover the same row. This is due to some active gene that forced the inclusion of a specific set which happened to cover a previously interpreted and active gene. We then take any set that covers some row already covered by another set, and check if all its elements are also covered by other sets. Whenever this happens, the set is dropped from the cover. Once the process is finished, the cost is worked out by adding together the costs of all sets still in the cover.

To understand what is going on, let us consider our sample problem in Table 3–1 and the situation depicted in Figure 3–5. Interpreting the genes sequentially from left to right, only genes [1, 3, 5, 6] would be active, corresponding to the cover $\{1, 3, 4, 5\}$. Careful examination shows that rows [1, 2] are covered by sets $\{1, 4\}$ and rows [3, 4] are covered by sets $\{3, 5\}$. So a redundancy check has to be carried out. It turns out that all rows in set 1 are covered by set 4 and all of those in set 3 also are in set 5, rendering sets $\{1, 3\}$ redundant. So the final cover would only be $\{4, 5\}$ and the cost is 40.

The information on which sets are active could be used in the future to guide the mutation operator somehow, concentrating thus the effort on those genes that are effectively affecting the cost.

3.5 Convergence for the Divide and Conquer Approach: Finding the Optimum

Premature convergence is a problem often found in classic genetic algorithm implementations. The proposed scheme, designed precisely to avoid this, has a certain noisiness in the evaluation function which prevents it from really converging. This is due to the re-emergence of new genetic material. The level of co-adaptation between the pools determines which individuals are fitter, and therefore makes sharper transitions in the individuals' fitness possible. The mutation rate influences to a great extent the process, allowing more or less material to reappear in the population. Small mutation values allow a kind of convergence, for which the algorithm cannot proceed any further, even though they show different alleles within the population.

The optimal solution is located by having the evaluation step check the score of the newly created combination against the previously best found solution, and recording any such combinations that have better scores. In this way, after a number of problem-determined generations, the execution of the algorithm is finished, and the best found solution is taken to be the optimum. Another plausible scheme would use the divide and conquer approach to generate a promising initial population for a classical genetic algorithm, which would thus start with a good population, that nevertheless maintains great variety in the genetic material, making it ideal.

3.6 Score assignment selection

The population is evaluated by sequentially picking each chromosome, along with peers from other pools. Jointly they form a complete solution of the problem, which is tested. The obtained fitness score is then assigned to each of them. All chromosomes are guaranteed to be evaluated at least twice, although they may be selected to evaluate individuals from other pools, in turn forcing more evaluations. The way the chromosomes are selected for evaluation can either be random or according to their last recorded fitness. As described above, initial testing showed that the *scored selection* tended to converge sooner than the *random selection* although often producing poorer quality solutions. Therefore *random selection* was chosen for the rest of the experiments. The reason seems to be that the selection pressure is incremented towards the best scoring individuals, because all chromosomes are tested on how well they score with the best peers found so far.

A change from either of those schemes to the other wasn't implemented as a command line option, but rather at compilation time.

3.7 Initialisation and Pool Assignment

The initialisation process is in charge of creating the data structures that will hold the problem information. The first decision to be taken is how the different rows are assigned to the pools.

As we know, the building block hypothesis is based on the assumption that interaction between genes, sometimes called epistasis[1], becomes smaller as the blocks grow. This means that promoting smaller building blocks will enhance the solution. Specifically, for the divide and conquer approach it seems important that relevant information is kept together in the same chromosome, so that the algorithms can evolve the chromosomes to produce better solutions. The set covering problem is used for bus driver scheduling problems for which the sets that cover a certain row are grouped together in the problem description file, as shown in Figure 3–6. We can see that for any of the sets contiguous rows tend to be covered more or less by the same set. Thus it seems that an ordered assignment of those rows to the pools will produce building blocks which the genetic algorithm will be able to optimise more easily. To test this, several assignment schemes were devised. The program allowed both the number of subproblems selected, i.e., the number of pools, and the element assignment strategy to be defined differently for each run. Initially the proposed strategies were:

[1] *epistasis* is used in genetics to refer to the suppression by a gene of the effect of some other that is not one of its alleles, [Makins 91].

Rows	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
1	1	1	1					1	1	1
2	1	1							1	1
3			1	1	1	1				
4	1			1	1	1				
5			1	1	1	1		1	1	1
6			1	1			1		1	1
7		1	1	1			1	1		
8		1	1			1	1	1		
9	1	1	1			1	1	1	1	
10		1	1	1			1	1	1	1

Figure 3–6: Set Covering Problem Matrix for Bus Driver Scheduling

- Random assignment of rows to each pool of equal size.

- Tournament assignment using some evaluation for the tournament, such as the inverse of the ratio of number of sets that cover this element, divided by number of total elements in the pool that those sets cover. We think this should tend to select those elements that are covered by the smallest number of subsets. This could reduce the number of subsets one has to work with in order to evaluate the solutions, thus reducing the size of the problem.

- Just a purely sequential assignment, i.e., grouping the elements in as many pools as wanted, using the ordering defined in the data file.

The first and third of these approaches were implemented. It remains to be tried out whether the second approach produces some improvement. Another variation in the sequential assignment would take into account whether neighbouring sets are closely related, and use that information to produce variable-sized subproblem pools.

In order to improve the quality of the initial populations the algorithm implements a *tournament initialisation* scheme[2] that tries to find a good coverer for each row. This is done by randomly selecting a number of sets that cover that element and assigning the best, i.e., the least expensive coverer, to the gene. The number is determined by a command-line parameter, and we will later show that this exerts some influence on the effectiveness of the genetic algorithm.

3.8 Other parameters in the Divide and Conquer Genetic Algorithm

3.8.1 Mutation

The mutation schemes described in chapter 2 can be implemented to affect just one gene, or to apply to each gene individually. Of course, the more genes involved, the greater the impact of this operator. Specifically, for **DCSCPGA** two different solutions were implemented. A command-line parameter is used to switch between the two. They constitute the possible extremes, with mutation only affecting at most one gene in one, or allowing it to affect all genes at once in the other.

3.8.2 Measure of Fitness

As described in section 2.2, the solution of the set covering problem involves minimising a function, subject to certain constraints. **DCSPGA** was designed to maximise a function. Therefore, the fitness of the individual had to be calculated based on the equation 3.1, which will reach its maximum when the cost is minimal.

[2]The idea is originally from Dave Corne.

We add one to the denominator to prevent unexpected division-by-zero errors.

$$Fitness = \frac{1}{1 + Cost\_of\_Cover} \tag{3.1}$$

The normal evaluation function is selected using the command line parameter `-escp<n>`, where n is the selection tournament.

3.8.2.1 Bus Driver Scheduling

Some of the real-world problems modelled by the set covering problem consist in the assignment of drivers in bus services. The cost associated with the cover is not the only parameter to minimise, but in fact it is more important to minimise the number of sets, which represent the number of drivers to cover the service. Therefore, the fitness function is modified to be

$$Fitness = \frac{5}{Number\_of\_Sets} + \frac{1}{1 + Cost\_of\_Cover} \tag{3.2}$$

The numerators act as weights that make the number of sets more important (in fact by a factor of five) than the cost of the cover. This weighting was decided in view of the costs and sets encountered in the data used in the experiments, but is highly dependent on their absolute values, which need to be taken into account to be of any use.

In this case the evaluation function is selected using the command-line parameter `-ebds<n>`, where again n represents the selection tournament size.

3.9 Results: The Output of the Program

3.9.1 Which are the results?

DCSCPGA, as did PGA, can both be run interactively or not. Depending on the option selected different results will be recorded. There are, nevertheless, several things recorded independently of the running mode selected. As described in section 3.3.3, the chromosomes are evaluated by combining together one from each pool, thus forming a complete solution of the problem. This is the only way in which the algorithm produces complete solutions. If it is to produce useful solutions, it needs to keep track of the best combined solutions found so far. This is done by keeping a file, named after the supplied command-line output file name, using the suffix `.cov` or, if none is provided, the default name `cover.cov` is used. The information included in this file is composed of four pieces of information:

Score This measure compares the cover about to be recorded with the last cover found. The value is given by

$$Score = \frac{\mathcal{CARDINAL}\{Union\} - \mathcal{CARDINAL}\{Intersection\}}{\mathcal{CARDINAL}\{Union\}} \tag{3.3}$$

It has been devised to produce one when both covers are completely different, and zero when they are the same. It therefore measures how similar those covers are, and is a key element used to decide if the new cover is to be recorded or not. In some way, this measure gives an idea of the amount of different genetic material found in the population. The greater that amount, the more varied the covers found and therefore the greater that measure.

Cost This keeps record of the cost associated with the cover. In straightforward set covering problems only those covers are recorded which either have a smaller cost than the previous one, or the same cost if the score is greater than zero. This prevents the program from repeatedly recording the same "best" cover.

Sets Under this heading the number of sets involved in the cover is stored. This information is essential in bus driver scheduling problems and is therefore kept here.

Cover The last piece of information is the cover found by the algorithm itself. The information stored displays the sets in that cover in ascending order.

Figure 3–7 shows an extract of the output of an actual run of the algorithm with one of the test problems described in section 4.2.1.

```
Score: 0.0735294        Cost: 430       Sets: 66
0 1 2 4 5 7 8 9 10 11 12 13 14 15 16 17 19 20 22 24 25 27 28 33
42 43 45 46 47 48 49 51 53 57 58 62 65 68 69 70 74 76 77 80 82 84
85 88 90 93 102 106 109 115 119 120 121 123 137 142 143 145 152
193 274 432

Score: 0.0735294        Cost: 429       Sets: 65
0 1 2 4 5 7 8 9 10 11 12 13 14 15 16 17 20 21 22 24 25 27 28 42
43 45 46 47 48 49 51 53 57 58 62 65 68 69 70 74 76 77 80 82 84
85 88 90 93 102 106 109 115 119 120 121 123 137 143 145 152 168
193 274 432
```

Figure 3–7: Extract from the Cover File

Inherited from PGA, the program is also able to create a file with the current chromosomes in the population. Its name is generated in a similar way, either by adding the extension `.chr` to the output file name in the command-line, or using the default `unnamed.chr`. The file contains both the fitness of the chromosome and the coding. They are sorted in descending order, and sequentially pool by pool. This information could be used to reconstruct a solution, or even to produce the initial population of a conventional genetic algorithm that would solve the set covering problem having taken advantage of the greater virtual population in the first stage. This would provide a highly fit though diverse initial population, and should therefore produce high quality solutions at a great speed.

3.9.2 Interactive versus Non-Interactive Running

On top of the information already described, **DCSCPGA**, as did PGA, can run both interactively and non-interactively.

```
(A)gain, (Q)uit, (C)ontinue: continuing..

        Divide and Conquer Genetic Algorithm: scp41.dat
     Subproblem Pools: 5       Chromosomes per Pool: 50
Generations per stage: 1000            Reproduction: simple_steady
   Reporting interval: 50            Crossover type: two-point
                                     Crossover rate: n/a
        Eval function: dc_sc         Mutation: single_bit 0.020000
            Selection: tn3
       Selection bias: n/a                  Generation: 600
                                  Evaluations so far: 3300
              Pool.......Average..........Best.(Min = unknown)
              0         449.0594603      447.0000000
              1         448.8964256      447.0000000
              2         449.0370296      447.0000000
              3         449.0350143      447.7462354
              4         448.8754696      447.0000000
```

Figure 3–8: Interactive Run of **DCSCPGA**

If running in the first mode, the program output is similar to that depicted in Figure 3–8. Different pieces of information are presented here. The title line prints the name of the data file used. Both the number of subproblem pools used and the number of chromosomes per pool are output in the next line. The third line displays the number of generations in a so-called stage, i.e., the number of generations that the genetic algorithm will evolve before the user is prompted for possible action. Next comes the reporting interval, i.e., the number of generations till the fitness values displayed below are refreshed, and information on the crossover operator being used. Then the name of the evaluation function is presented (this can either be set covering or bus driver scheduling), the mutation operator, the selection scheme, generations, evaluations, and lastly the pools with their average scores and best found scores.

```
=====================================
Divide and Conquer Genetic Algorithm
            data_file = scp41.dat
                 eval = dc_scp
          max-fitness = unknown
         reproduction = simple_steady
               select = tn3
       selection-bias = n/a
                Pools = 5
Chromosome # per pool = 50
       crossover-type = two-point
       crossover-rate = n/a
        mutation-rate = 0.020000
   reporting-interval = 50
   random-number-seed = 810142742


Generations = 0  Evaluations-so-far = 0
  0        585.4936697      535.8362627
  1        587.1880796      539.3394834
  2        587.8372163      546.7161332
  3        583.1582582      537.5616311
  4        589.2917679      539.1582284
```

Figure 3–9: Non-Interactive Run of **DCSCPGA**

If running non-interactively, **DCSCPGA** will produce an output file. The information in this file will have a small header with the options used to run the program, and will save each reporting interval, as well as the average and fittest solution scores in the pools. An extract of such a file can be seen in Figure 3–9.

Chapter 4

Experiments

4.1 Description of Objectives

The main objective in the testing phase was to establish the impact that the divide and conquer approach produced in the results. This meant having to establish first the "best" general settings in which the experiments had to be carried out, and then comparing the results obtained for different values of

- Number of Pools
- Pool Assignment Scheme
- Influence of Mutation

4.1.1 Effect of the Number of Pools

The basic idea in the proposed approach is to create a set of solution pools for subproblems derived from the original problem. In order to test and compare the results, a set of collections of ten experiments were devised using different numbers of pools. These problems are described in section 4.2.

Problem	Pools								
SCP41	1	5	10	15	20	25	30	35	40
SCP42	1	5	10	15	20	25	30	35	40
SCP43	1	5	10	15	20	25	30	35	40
BDSP1	1	5	10	15	20	25	30		
BDSP6	1	5	10	15	20	25	30		
BDSP7	1	5	10	15	20	25	30		

Table 4–1: Experimenting with the Pool Number

4.1.2 Effect of the Pool Assignment Scheme

As discussed in section 3.7, the ordering in the pools may be an important factor in the solution of bus driver scheduling problems. To test this hypothesis, two sets of groups of ten experiments were carried out on each of the bus driver scheduling problems described below in section 4.2.2, one where the rows are assigned randomly to the pools, and another where they are assigned sequentially.

4.1.3 Mutation Schemes and its importance

The last element to determine is the influence of mutation on the solution procedure. As we know, mutation is the only operator capable of producing new material and incorporating it into the population.

Mutation's main mission is to guarantee that the enviroment around good individuals is explored thoroughly, so that the minima are located and the algorithm is prevented from getting stuck at local optima. The divide and conquer approach tries to mantain variety in order to locate all maxima. But if a great deal of mutation is needed, this seems to suggest that the population has converged too much, and it is up to mutation to take the algorithm out of the deadlock. It has to be kept in mind, though, that using small populations means that the approach cannot be as effective. The size has to be fixed small enough though to allow time consuming testing to be carried out.

4.2 Description of the Test Problems

In order to test the divide and conquer idea, seven different problems were used, three of which were randomly generated set covering problems, while another three were real-world bus driver scheduling problems, the last being a randomly created one.

The size of the problems selected is small to allow faster testing of the algorithm. Population size and resources were accordingly kept small, except in those tests in which the parameters tested required otherwise.

4.2.1 Operations Research Library Problems

The Operations Research Library is an on-line open accessible collection of operations research problems maintained at the Imperial College in London and described in [Beasley 90]. The idea behind having gathered such a collection of standard problems is that "algorithms can be compared on a more realistic basis (with regard to their performance on exactly the same set of test problems) than would otherwise be so" if not publicly available.

Specifically, the Operations Research Library keeps several standard set covering problems. Three of those, originally from [Balas & Ho 80], were selected to test the performance of the divide and conquer approach.

The problems used were originally randomly generated aiming at a specific density. The characteristics of the problems are

Problem	Rows	Columns	Density	Optimal Cost
4.1	200	1000	2%	429
4.2	200	1000	2%	512
4.3	200	1000	2%	516

Table 4–2: Operations Research Library Problems

Appendix B has sample optimal covers for those problems provided by Paul C. H. Chu. These problems will be referred to as Scp4x, where x is the problem number, throughout the rest of this document.

4.2.2 Bus Driver Scheduling Problems

The problems used to test the approach applied to bus driver scheduling are provided by Anthony Wren, from the Scheduling and Constraint Management Group of the University of Leeds. As we know, bus driver scheduling tries to reduce the number of sets or shifts used to cover the service. Therefore the parameter to optimise is the number of sets instead of the cost.

Problem	Rows	Columns	Optimal number of Shifts
1	200	539	Unknown
6	422	3082	37
7	216	1058	19

Table 4–3: Bus Driver Scheduling Problems

The details of the bus driver scheduling problems can be seen in Table 4–3. These problems will be referred to as Bdspx, where x is the problem number, throughout the rest of this document.

4.2.3 Randomly Created Problem

In order to test the influence of mutation on the results, a small random problem was developed. This problem was created using the program *create* described in appendix A.

The characteristics of the problem itself are

Rows	Columns	Density	Best Known Solution
30	90	20%	21

Table 4–4: Randomly Created Set Covering Problem

4.3 Testing Plan

This section describes the specific testing plan followed. First of all, problem Scp41 was used to determine the effectiveness of the pool number by decoupling it from the effect of a high mutation rate and a favourable initialisation tournament, while the program was tested on a problem with known solution.

Once this was done, a set of organized experiments were carried out on each problem. Firstly, using a fixed number of pools, five, and *Multi-Gene* mutation with mutation rate of 0.35, the optimal value of the initialisation tournament was established for the set covering problems of the *Operations Research Library* using scp41. The experiments tried a range of initialisation tournament sizes. With the results of those, the best value was selected and applied to all set covering problems. The number of pools was then studied. This included experimenting with a whole range of pool numbers. The precise experiments are pictured in Table 4–5.

As to the bus driver scheduling problems, an analogous testing plan was developed that first tried to establish the ideal initialisation tournament value, and

Problems	Initialisation Tournament	Pool Number	Mutation
Scp41	1	5	0.25
Scp41	10	5	0.25
Scp41	20	5	0.25
Scp41	30	5	0.25
Scp41	40	5	0.25
Scp41	50	5	0.25
Scp41, Scp42, Scp43	40	1	0.3
Scp41, Scp42, Scp43	40	5	0.3
Scp41, Scp42, Scp43	40	10	0.3
Scp41, Scp42, Scp43	40	15	0.3
Scp41, Scp42, Scp43	40	20	0.3
Scp41, Scp42, Scp43	40	25	0.3
Scp41, Scp42, Scp43	40	30	0.3
Scp41, Scp42, Scp43	40	35	0.3
Scp41, Scp42, Scp43	40	40	0.3

Table 4–5: Experiments on the Set Covering Problems in the Operations Research Library

then, (given that value) determined the effect of having different pool numbers. The corresponding schedule is displayed in Table 4–6.

Lastly, some experiments were carried out on a small randomly created problem, in order to be able to determine the influence of the mutation rate on the population scores. This experiments used both different pool sizes and variable mutation rates.

Problems	Initialisation Tournament	Pool Number	Mutation
Bdsp7	1	5	0.25
Bdsp7	5	5	0.25
Bdsp7	10	5	0.25
Bdsp7	15	5	0.25
Bdsp7	20	5	0.25
Bdsp7	25	5	0.25
Bdsp1, Bdsp6, Bdsp7	10	1	0.3
Bdsp1, Bdsp6, Bdsp7	10	5	0.3
Bdsp1, Bdsp6, Bdsp7	10	10	0.3
Bdsp1, Bdsp6, Bdsp7	10	15	0.3
Bdsp1, Bdsp6, Bdsp7	10	20	0.3
Bdsp1, Bdsp6, Bdsp7	10	25	0.3
Bdsp1, Bdsp6, Bdsp7	10	30	0.3

Table 4–6: Experiments on the Set Covering Problems in the Operations Research Library

Chapter 5

Preliminary Experiments

5.1 General Comments

The first experiments carried out in order to fine tune the parameter values and to verify that the program was actually working were performed using test problem Scp41 of the Operations Research Library.

This set of experiments studies in detail both the evolution of the pools and the "best covers" found. Different pool numbers were used in order to establish if the divide and conquer approach was actually producing any different or interesting results.

We know from section 4.2.1 that the optimal cover for this test problem has an associated cost of 429. Although this optimum has only been found once during the testing phase of the project, nonetheless, it does not undermine the approach itself, but if anything, points to the need to improve the genetic algorithm run. As we will see in this and the next chapter, a comparison of the results obtained using a traditional genetic algorithm with the ones produced through our approach, reveals an important increase in performance in the latter. Nevertheless, the optimum value has to be kept in mind as the lower boundary when evaluating the results of the experiments.

The statistics have been calculated on the basis of the best scores recorded for each pool after ten different runs. The difference with what would have been obtained using the average scores is negligible in all, but the experiments involving fifty pools. The reason for the special behaviour of that case will be dealt with specifically in the corresponding section.

These experiments were carried out using single gene mutation and a low initialisation tournament. This means that the only real source of good genetic material was recombination from what was still present in the population. The small difference between best and average scores in the pools at the end of each run suggests that towards the end the algorithm has lost much of the variation in the material originally present, and that therefore it should need to rely more heavily on mutation. This observation agrees with the results reported in [Beasley & Chu 95]. They suggest that "in order for the GA to be effective, a higher mutation rate is necessary when the GA has converged. (...) it is beneficial to utilise a variable mutation rate rather than a fixed one and this variable rate should depend on the rate at which the GA converges". The rest of the experiments carried out in this work, as reported in chapter 6, used multi-gene mutation and high rates. This turned out to improve the scores independently of the number of pools used.

5.2 Detailed Experiments

To investigate the performance of the GA in terms of the best quality solutions produced, several experiments were carried out for both one and twenty-five pools.

The results of those experiments mimicking traditional genetic algorithms, i.e., those using a single pool, can be seen in Figure 5–1, which shows the evolution of the best cover recorded by the algorithm as it ran. The populations for this set of experiments were allowed to evolve some five thousand generations.

It can be seen how the cost reaches some point where a saturation effect is observed. It seems that the algorithm is unable to improve any further. The graph

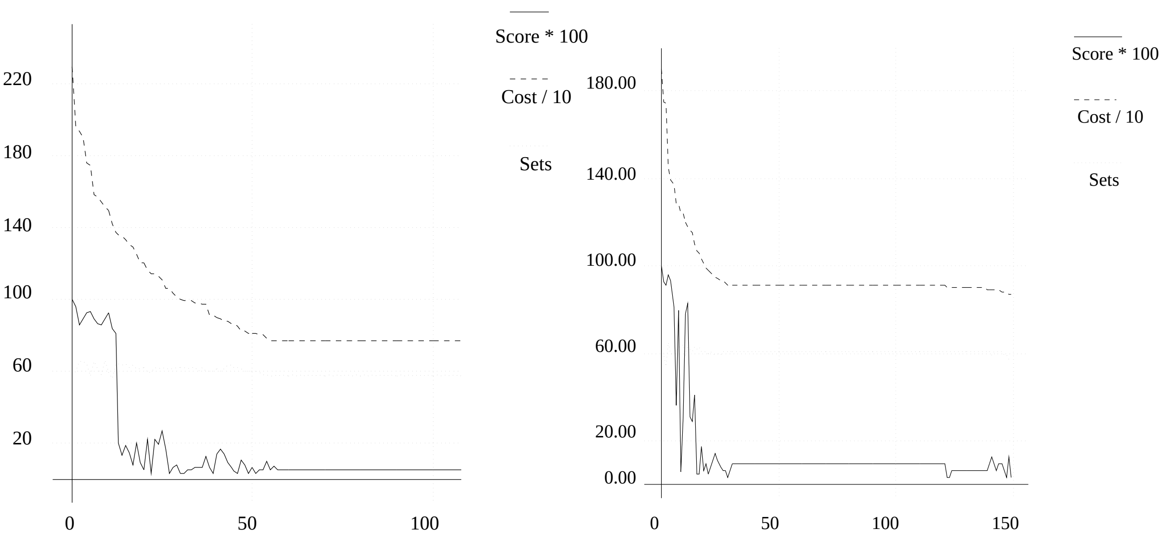

Figure 5–1: Cover Evolution for one pool (50 and 300 Individuals)

on the right-hand side corresponds to an experiment for which the population size was set to 300 individuals. As would be expected, it takes longer for the genetic algorithm to produce acceptable results, but even after 7500 generations, the algorithm seems unable to improve the costs any further. The improvement observed towards the end seems to be caused by mutation, which further suggests that this operator is of crucial importance.

The evolution of the score measure described in section 3.9 suggests that a greater amount of genetic material is still in the population. The flat region corresponds to a zone for which the algorithm is alternatively generating two solutions of equal cost. The score is maintained at the same level; thus the algorithm is just finding two sets of three columns which are exchanged, producing equivalent covers.

Figure 5–2, on the other hand, shows how the *"best-found-cover"* evolves in the case of twenty-five pools. Here the score measure seems to drop smoothly as the cost decreases. Then, after 5000 generations, the algorithm once again gets stuck in some local minimum and is unable to proceed any further on to better quality solutions, and only mutation is able to carry the search further.

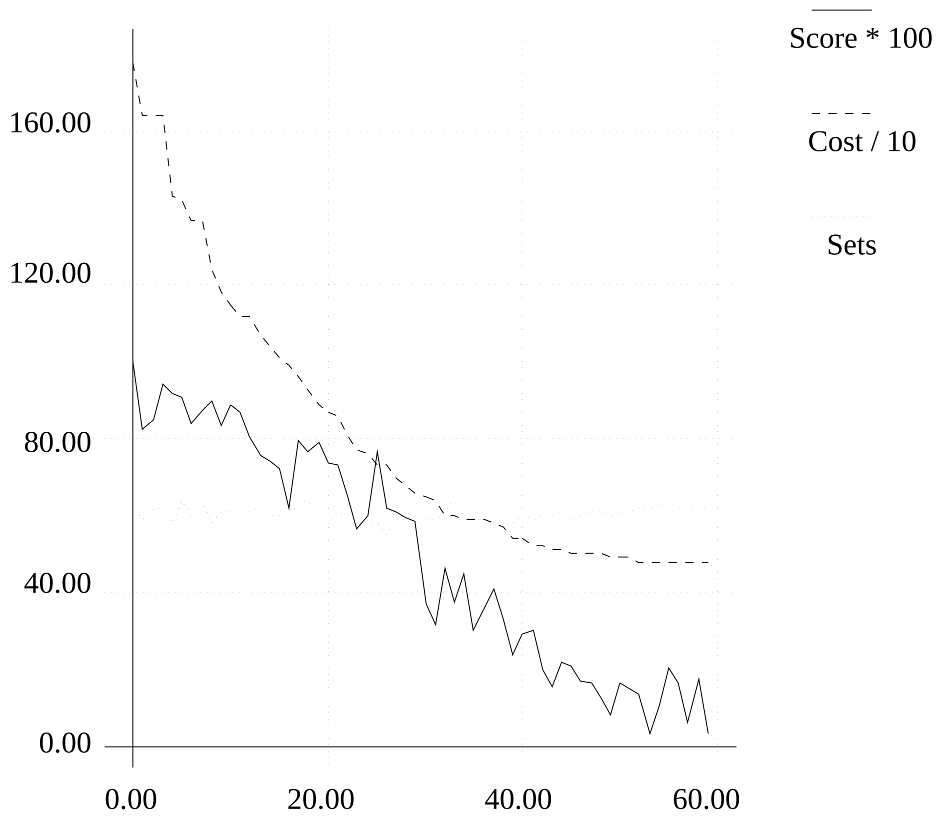

Figure 5–2: Cover Evolution for 25 pools

It can be clearly seen that improving the number of pools has had a positive effect on the evolution process, allowing better solutions to be found. It seems, though, that mutation is needed in order for the genetic algorithm to find even better solutions. As a proof of this, Figure 5–3 shows the results of a run using eighteen pools, multi-gene mutation, and a mutation rate of 0.2. The algorithm produced 3000 generations, after which the optimum had already been found.

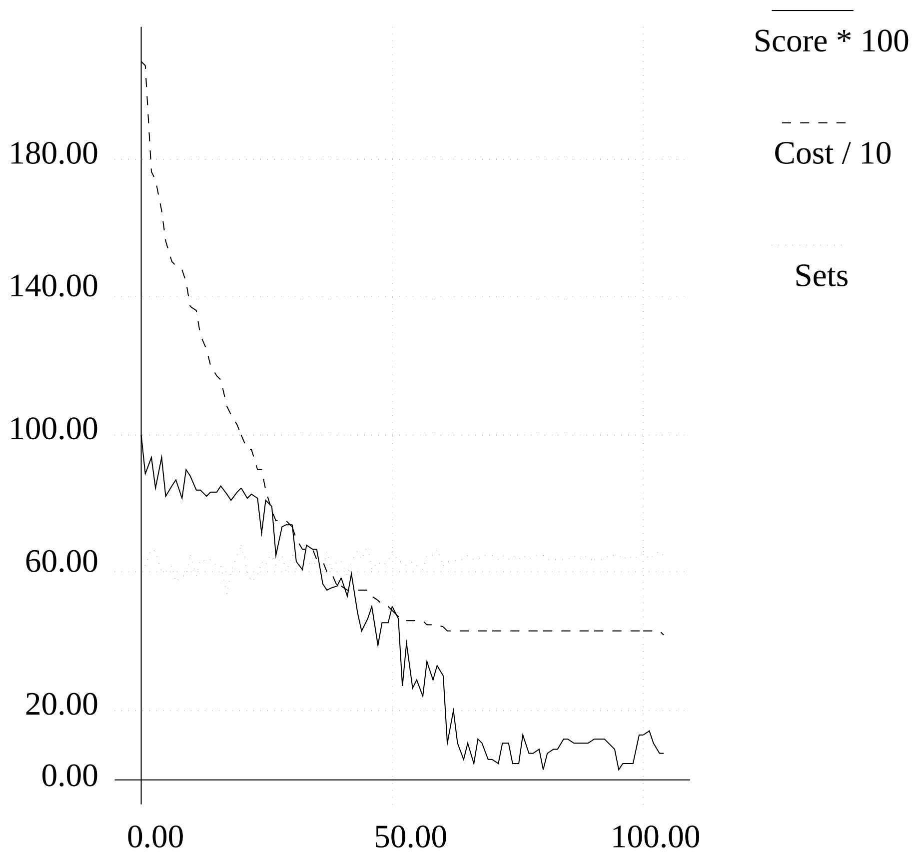

Figure 5–3: Cover Evolution using Multi-Gene mutation

5.3 Examining the final pool Scores

5.3.1 Experiments with One Pool

The results of the experiments using one pool can be seen in Table C–1. Although the best, i.e., the minimum score for this problems is (as we already know) 429, the algorithm is totally incapable of even getting near to it. The problem here is originated by premature convergence. The population size is too small, just 50 chromosomes, and having only one pool means that the amount of genetic material contained in the population is too small to be of any use. Increasing the size of the population seems to improve the quality, but a six-fold increase of the population made only a small difference in the quality of the final scores, while requiring a greater time to complete.

5.3.2 Experiments with Five Pools

The use of five pools, as shown in Table C–2 seems to produce a better behaviour compared to just one pool . Even a small number of pools already makes a significant impact on the results of the genetic algorithm. This is consistent with the results reported throughout chapter 6.

The standard deviation is reduced, and this means that the solutions tend to fall within a very small interval around the average cost shown. In other words, the different runs did converge to almost the same values, showing a more robust behaviour.

One further thing that can be seen is that, comparing adaptive versus non-adaptive mutation, the latter seems to produce better quality of final solutions when all other parameters remain the same.

5.3.3 Experiments with Fifteen Pools

Increasing the pool number to fifteen already nearly produces the best results the algorithm is capable of. The costs reported in Tables C–3 and C–4 are similar to those obtained using twenty-five pools. The optimum performance probably requires some pool size between these two values.

It seems that those experiments that use non-adaptive mutation tend to yield smaller costs than those that implement adaptive mutation. Examining the evolution of the pools for different generation values, we can see that adaptive mutation has a faster convergence. The greater mutation value reached towards the end avoids the settling in the pool scores. This does not mean that better "best covers" cannot be found, but a certain kind of equilibrium is reached stabilising the pool scores: having to combine each chromosome with those in other pools in order to evaluate its fitness, and given that the final score is obtained by averaging several evaluations, the effect of the slightly greater mutation rate does not translate into variations in the individual scores.

As to the variation in the cost among the different pools, all mean values appear to be almost equal. This seems to indicate that small improvements are averaged out of the population, and it is left to mutation to further enhance the solutions.

5.3.4 Experiments with Twenty-five Pools

The results obtained using twenty five pools are reported in Tables C–5, C–6, C–7, and C–8. As said above, the costs are comparable to those produced by 15-pool genetic algorithms. The computational cost, though, is far bigger in this case because ten more pools mean (supposing half of the chromosomes have already been evaluated twice before they go onto evaluation themselves) that another 500 evaluation steps need to be carried out every five generations. All in all, the best overall scores were found here in the case of sequentially assigned rows with non-adaptive mutation. Those are nevertheless far from the optimum value of 429.

Again there is but a small difference in the mean scores obtained for the different pools, i.e. the genetic material in the population seems to have converged to a point where no further enhancements can be produced without the aid of mutation.

5.3.5 Experiments with Fifty Pools

The results obtained using fifty pools seem to suggest that there is a limit to the amount of pools that can be used. In this case the size of the chromosomes is $Size\_of\_problem/Number\_of\_Pools = 200/50 = 4$.

This means that the size of the chromosome is far to small for it to be of any use. The rationale of the argument is that the chromosomes become good solutions if they tend to cover more sets with less cost, but being so small, some of them might be completely inactive, i.e., all the rows are already covered by some sets that cover active rows in other pools, and so they do not contribute to the final score. They may not be evaluated at all. This means that they do not improve with time. Should they become active at some later time in an offspring, the new chromosome will not be as fit as the rest, and it dies out. So, although it may well be that one gene within that chromosome is vital in producing a greater quality solution, it is never included in the evaluation stage, and never gets the chance to evolve towards a better allele.

The search carried out here is therefore blind, and even after 3500 generations the algorithm seems unable to produce any good quality individuals. The big values in the standard deviation further suggest that the search is not being sufficiently guided. The costs depend highly on the initial population.

The results of the batch runs can be seen in Tables C–9, C–11, C–13, and C–15. We can see that the differences in the scores among different pools is still great, especially in terms of the best individual scores. This strengthens the argument that some pools have not really had the opportunity to improve.

Chapter 6

Results

6.1 Problems in the Operations Research Library

6.1.1 Determining the Initialisation Tournament Size

Tournament Size	μ_{Cost}	σ_{Cost}	μ_{Time}	σ_{Time}
1	480.90	16.43	63.00	1.41
10	469.80	10.82	63.30	1.89
20	453.00	7.87	63.80	2.39
30	446.67	3.87	64.44	1.13
40	439.20	2.70	64.70	2.41
50	441.00	2.94	63.70	1.34

Table 6–1: Problem Scp41: Experiments with variable Initialisation Tournament Sizes

As described in section 4.3, the first experiments were intended to determine the optimal size of the initialisation tournament. This tournament, described in section 3.7, is intended to heuristically produce a good initial population that attains better quality solutions in less time. The experiments allowed the algorithm to evolve for three thousand generations. The results are shown in Table 6–1. It can be clearly seen that a big initialisation tournament size produces a better result in less generations, even though only five pools have been used. The best result was obtained using a size of forty, which also has the smallest observed standard deviation. This was the value used in all further experiments that use the set covering problems obtained from the Operations Research Library.

6.1.2 Efficiency of the system: Changing the Number of Pools

Pools	μ_{Cost}	σ_{Cost}	μ_{Time}	σ_{Time}
1	495.70	13.76	412.86	5.46
5	447.90	3.38	873.24	9.94
10	445.70	4.90	1448.97	183.83
15	441.50	2.95	1910.04	10.43
20	437.90	1.79	2538.27	35.57
25	436.20	1.03	3165.03	34.04
30	434.60	1.35	3594.63	29.80
35	434.60	1.35	4246.29	39.52
40	434.50	1.18	5330.47	50.85

Table 6–2: Problem Scp41: Experiments with different Pool Sizes

Once the optimal value for the initialisation tournament was determined, the next step was to try and establish the influence of the pool number in the final result, and to check how good a solution the divide and conquer approach provides to maintaining variety. The experiments using problems Scp41, Scp42 and Scp43 can be seen respectively in tables 6–2, 6–3, and 6–4. All experiments for the first two problems, as well as those involving 35 and 40 pools in the case of the

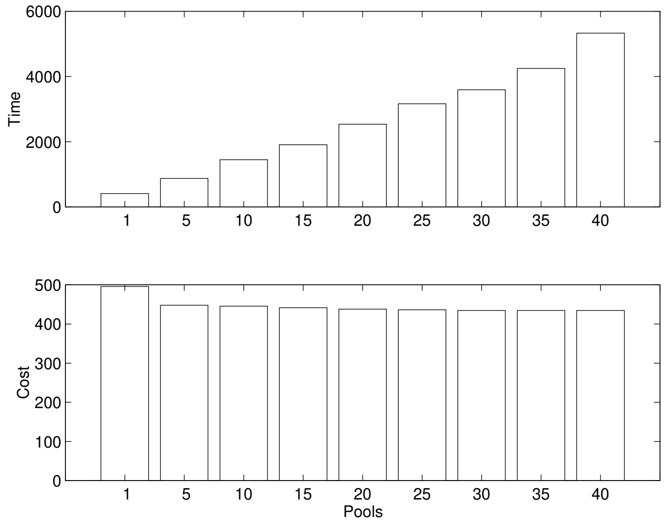

Figure 6–1: Scp41 Comparing time efficiency and costs

third problem, were carried out on the same machine. The rest of the experiments for that problem were carried out on another machine. Therefore, the times have to be compared among themselves, and are not comparable with others.

There is a clear improvement in the results of the algorithms as the number of pools increases. The increase is most dramatic when comparing the solution using a single pool, which corresponds to a traditional genetic algorithm, with the use of only five pools. In relative terms, there are respective improvements of 9.64%, 7.82%, and 7.93% associated with just such a small number of pools. The standard deviation is also notoriously smaller, which means that the algorithm is in some way more robust, and finds better quality solutions more often. As the number of pools increases, more improvements can be seen, although the effect gets smaller, and the relative increase in efficiency is not so spectacular. It must be underlined, though, that whatever the problem, increasing the number of subproblem pools resulted in better performance in all but two cases[1].

[1] Those two are 35 pools in both Scp42 and Scp43.

Pools	μ_{Cost}	σ_{Cost}	μ_{Time}	σ_{Time}
1	595.60	10.84	429.41	15.52
5	549.00	5.40	914.89	75.89
10	548.30	6.40	1491.64	138.37
15	542.40	4.35	1963.42	22.40
20	537.00	6.67	2617.40	131.74
25	533.90	4.23	3260.81	79.02
30	531.60	4.06	3651.69	18.99
35	532.50	5.80	4268.44	38.29
40	525.40	5.21	5452.41	62.77

Table 6–3: Problem Scp42: Experiments with different Pool Sizes

Time is also an important factor in judging the overall performance, although the results show that, at least for all the problems, the approach does pay off in solution quality. Nonetheless, it has to be kept in mind that using a greater number of pools means increased computational complexity, and therefore greater processing time. Figures 6–1, 6–2, and 6–3 relate the mean cost and processor time for each number of pools. We can see that running the algorithm with one pool took nearly seven minutes of processor time[2], whereas a 40-pool-run would take abount one and a half hours. Therefore, it is not at all clear what the optimum number of pools to use is.

In the case of Scp41 there is not too much difference in the performance for 20, 25, 30, 35, or 40 pools. Especially in the last three cases, the results fell within one standard deviation of each other, making them practically the same. The computation time, on the other hand, ranged from about 45 minutes for 20 pools

[2]This is valid for Scp41 and Scp42

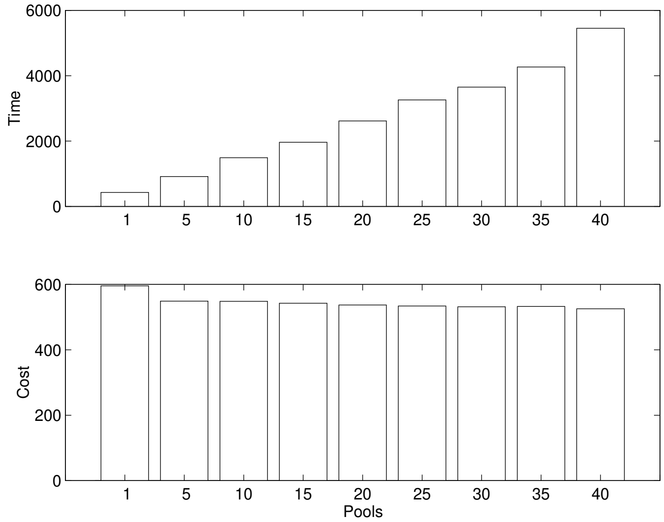

Figure 6–2: Scp42 Comparing time efficiency and costs

to 90 minutes for 40 pools. This would place the optimum pool size at about 25 pools.

There is a similar behaviour for Scp42, where the increase in performance between 20 and 40 pools is a mere 2.16%, yet the amount of processor time is increased by a 208%. Again, to ensure efficiency, the compromise between pool number and speed lies around 25 pools.

Although run on a different machine, we can observe the same trend in the results of Scp43. These are approximately the same between 20 and 40 pools, where nearly all lie within one standard deviation of each other. Again it seems that 25 pools is a good compromise.

Using 25 pools means for all three problems that the chromosome size for each pool is:

$$ChromoSize = \frac{ProblemSize}{NumberofPools} = \frac{200}{25} = 8 \tag{6.1}$$

It seems that the algorithm works fine using chromosomes with a size between

Pools	μ_{Cost}	σ_{Cost}	μ_{Time}	σ_{Time}
1	612.90	5.65	1343.34	4.28
5	564.30	6.41	2864.39	15.04
10	559.50	5.62	4537.07	19.73
15	550.70	7.41	6301.58	30.53
20	540.40	6.93	8368.85	34.77
25	537.40	2.91	10510.27	59.94
30	535.60	4.48	11808.74	66.18
35	542.70	6.40	4221.81	24.06
40	527.10	1.73	5574.51	369.52

Table 6–4: Problem SCP43: Experiments with different Pool Sizes

five and ten, corresponding to 20 and 40 pools and eight, in this case is a fine compromise.

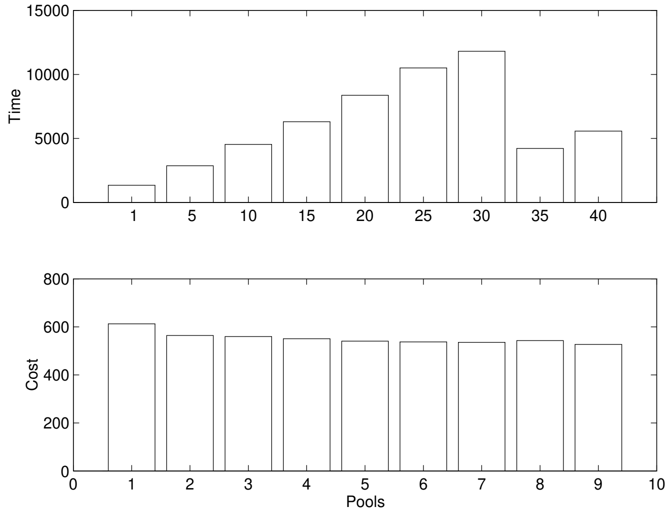

Figure 6–3: SCP43 Comparing time efficiency and costs

Note: The experiments for 35 and 40 pools were carried out on a different machine and the times are therefore not comparable

6.2 Bus Driver Scheduling Problems

6.2.1 Determining the Initialisation Tournament Size

Tournament Size	μ_{Sets}	σ_{Sets}	μ_{Cost}	σ_{Cost}
1	19.50	0.527	9722.9	129.05
5	19.90	0.316	9867.0	94.71
10	19.70	0.483	9797.7	135.51
15	19.70	0.483	9745.1	131.44
20	20.00	0.000	9837.4	55.70
25	19.80	0.422	9822.8	106.51

Table 6–5: Problem BDSP7: Experiments with variable Initialisation Tournament Sizes

In order to determine the optimal initialisation tournament size, a set of experiments, analogous to that described in section 6.1.1, was carried out for bus driver scheduling problems. In this case, the results did not show such a clear improvement as the tournament size was increased. It turns out that they are nearly the same no matter the size of the tournament used. This has to do with the fact that the actual covers involve only a small number of sets, whereas the problem used for testing, BDSP7, had a total of 1058 sets. The initialisation heuristic has too many sets to select from, and only a small number would produce a good initial population. Furthermore, only a few genes are active, and therefore only a small number of the sets selected will contribute towards the final score. Therefore, the heuristic is of little (if any) help in the search and it is left to the genetic algorithm to do all the work. So, regardless of the size of the tournament used, the result is approximately the same.

A size of ten was chosen for the rest of the experiments. A tournament size of one delivered greater standard deviation, and the results for five were worse than both one and ten.

6.2.2 Bus Driver Scheduling and Number of Pools

	Random Assignment		Sequential Assignment	
Pools	μ_{Sets}	σ_{Sets}	μ_{Sets}	σ_{Sets}
1	21.70	0.483	21.62	0.650
5	19.60	0.516	20.00	0.000
10	19.80	0.422	20.00	0.000
15	19.90	0.316	19.90	0.316
20	20.00	0.000	20.00	0.000
25	19.90	0.316	19.80	0.422
30	19.90	0.316	19.90	0.316

Table 6–6: Problem BDSP7: Experiment with different Pool Sizes

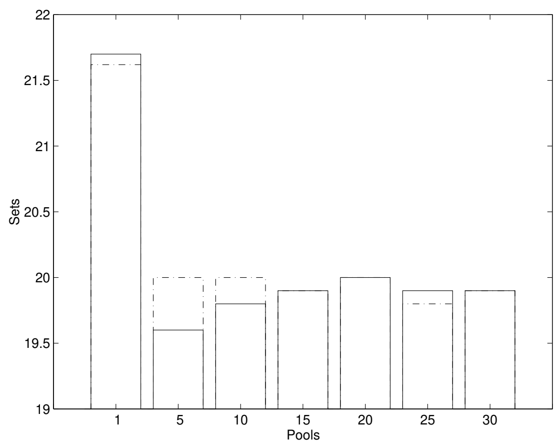

Figure 6–4: BDSP7 Costs (Solid: Random – Dash-dot: Sequential)

Pools	Random Assignment μ_{Sets}	σ_{Sets}	Sequential Assignment μ_{Sets}	σ_{Sets}
1	16.50	0.527	16.20	0.632
5	16.00	0.000	15.90	0.316
10	16.10	0.568	15.80	0.632
15	15.90	0.568	15.90	0.316
20	16.00	0.471	15.90	0.316
25	15.80	0.422	15.90	0.316
30	15.80	0.632	15.90	0.568

Table 6–7: Problem BDSP1: Experiment with different Pool Sizes

After having determined the initialisation tournament size, the next step was to try to determine the effect that the use of subproblem pools has on the performance of the genetic algorithm in the case of bus driver scheduling. Two sets of experiments were carried out as described in section 4.3, using both random and sequential assignments. The respective results can be seen in Tables 6–6, 6–7, and 6–8.

First of all, it has to be made clear that although the results are not so impressive as they were in the case of the Operations Research Library problems', the divide and conquer approach does pay off in that for all three test cases, whether or not sequential pool assignment was used, employing one pool would produce worse results that using five or more.

The nature of the first two problems makes the results less impressive than the ones observed in the straightforward set covering problems. The relative increase in performance associated with the use of five pools compared to the traditional genetic algorithm, as shown in Figures 6–4 and 6–5 is a mere 9.7% in the first problem for random assigned elements, and 7.5% if they are sequentially allocated to the pools. The results for BDSP1 are even less spectacular, amounting to a to-

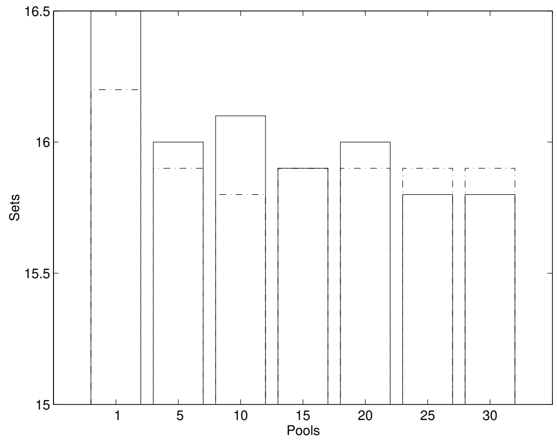

Figure 6–5: BDSP1 Costs (Solid: Random – Dash-dot: Sequential)

tal of 3% if random assignment was used, and just 1.8% for sequentially assigned experiments. These poor results reflect that the difficulty in solving those problems, given that solutions have a very small number of sets, does not lie in finding covers with a small, but in the fact that the fitness landscape is nearly flat, and a great amount of search corresponds to a little improvement. But although small, it takes the algorithm a great deal of effort, and the overall results are therefore encouraging.

If we consider instead problem BDSP6, depicted in Figure 6–6, for which the optimal solution has 39 sets, we see a better relative performance improvement record. Here, randomly created pools score 17.5%, whereas sequentially allocated pools reach 16.5%. This suggests that the expected performance improvement will increase with bigger problems.

The order seems to be of secondary importance. This, at least, is what the comparison of the results obtained using sequential and random assignment suggests. There is a slight improvement in problems using bigger chromosomes, where the chance of gene ordering being more influential is greater. This corresponds to a small pool number. The extreme exemplified in the traditional genetic algorithm always produced better results for the sequentially allocated problems although

Pools	Random Assignment μ_{Sets}	Random Assignment σ_{Sets}	Sequential Assignment μ_{Sets}	Sequential Assignment σ_{Sets}
1	48.70	0.483	47.80	1.033
5	40.20	0.789	39.90	0.876
10	39.40	0.699	39.40	0.516
15	39.40	0.699	39.30	0.483
20	39.30	0.483	39.40	0.699
25	39.40	0.699	39.30	0.823
30	39.50	0.850	39.80	0.632

Table 6–8: Problem Bdsp6: Experiment with different Pool Sizes

the values fell well within one standard deviation of each other. Furthermore, the effect is diminished by increasing the number of pools. This suggests that there is no better ordering in the allocation of rows to pools.

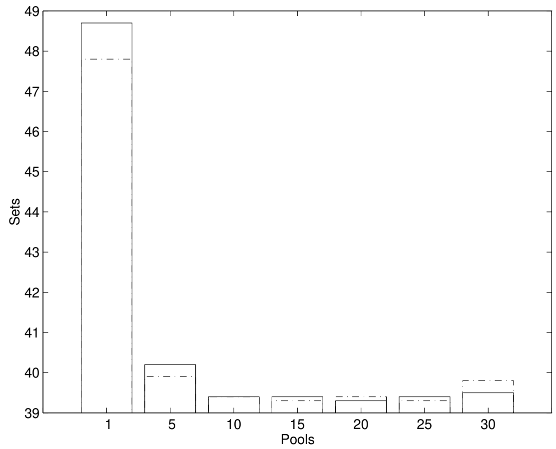

Figure 6–6: Bdsp6 Costs (Solid: Random – Dash-dot: Sequential)

6.3 The Random Problem generated with `create`

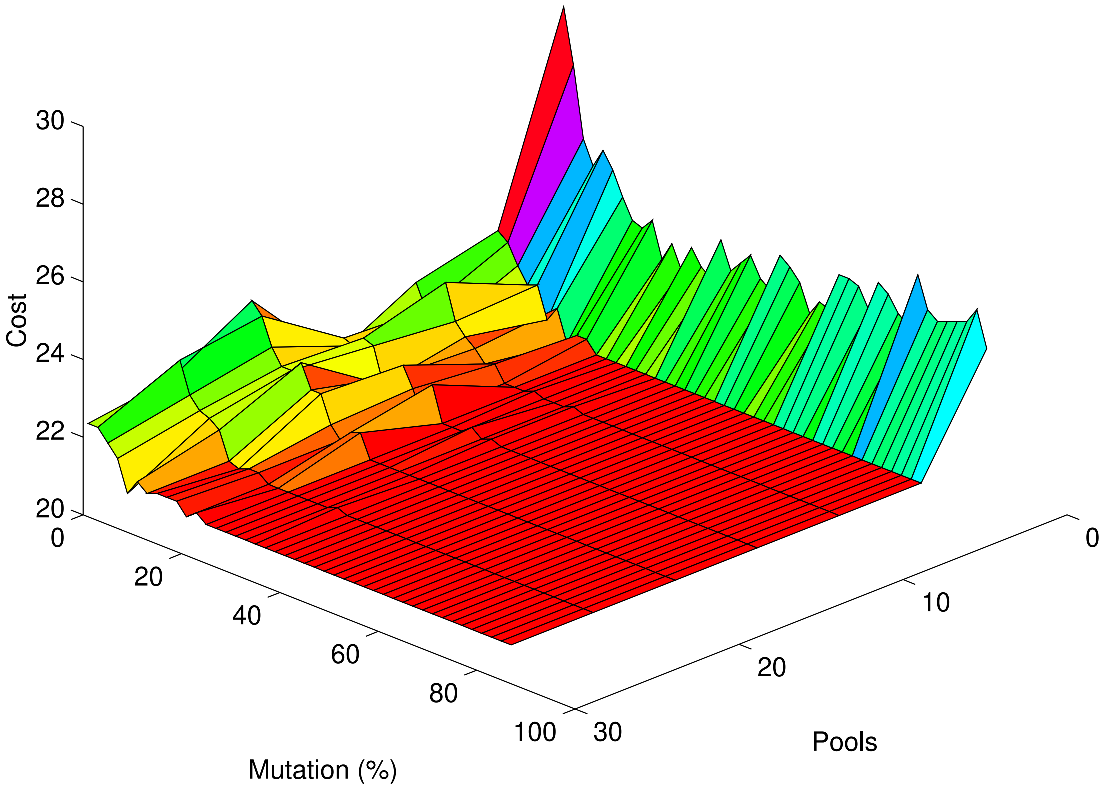

Figure 6–7: RANDOM PROBLEM Joint influence of Mutation and Pool Number

The random problem described in section 4.2.3 was especially created in order to have a test case on which to extensively investigate the joint effects of mutation and different pool sizes. In order to be able to produce enough measurements, the size of the problem needed to be small so that the algorithm would run fast enough. The exhaustive experiments were carried out with {1, 5, 10, 15, 20, 25, 30}, and using *Multi-Gene* mutation rates ranging from 1% to 87%. The results are represented in Figure 6–7.

The graph has several interesting features. First of all, it can be clearly seen that increasing the number of pools will in effect produce better results. It has to be kept in mind that the size of the problem makes it easy to solve it even with a small number of pools. This explains why the optimum is reached already with few pools if the mutation rate is big enough. Another interesting observation is that, as stated several times above, mutation is a crucial operator for the set covering problem. We can see that for any number of pools the algorithm performs better if a higher mutation rate is used. Of course, the greater the number of pools, the lower the critical mutation rate after which the genetic algorithms are capable of finding the optimum cover.

The algorithm seems to be more robust for greater mutation rates and number of pools. So, for the traditional genetic algorithm that uses a single pool, the performance deteriorates if the mutation rate is too high. This effect cannot be seen at all for greater pool numbers. Of course, the small size of the problem helps here, but nevertheless the effect is worth mentioning.

Lastly, we can see that having twenty pools in this case has the strange effect of producing worse results than either fifteen or twenty-five. This has to be attributed to some capricious characteristic of the problem.

One last comment on the phenomenon reported in section 5.3.5, by which the size of the chromosome seemed to impose a limit on the number of pools used: the results here suggest that this might not be true. But a closer look reveals that in this case it is due to the simplicity of the problem.

Chapter 7

Conclusions

7.1 Evaluating the Approach

There is a growing awareness of genetic algorithms, as well as other evolutionary techniques, in management problems, but we are only starting to uncover what their real strengths and caveats are. This work set out to try and determine the feasibility of the divide and conquer approach to solving a specific type of common operations research problem used to model many real-world interesting situations.

The approach was implemented and tested on a rather small set of standard problems, as well as on some real-world data. Extensive testing did point some facts out, although the limited amount of problems we were able to test due our time constraints has left several questions open.

It has been shown that there is a significant improvement in the solutions when compared to a traditional genetic algorithm that otherwise uses the same coding and evaluation function. This means that, in theory, improving either or both of these would immediately result in an improvement in the efficiency of the current program. Another observation has been that there is a whole range of pool sizes that seem to produce almost the same results. Of course, at least with the present evaluation scheme, a tiny increment in the number of pools can produce big differences in processor time, rendering the genetic algorithm infeasible compared to industry standard algorithms. These pool sizes that produce so little quality difference need to be further determined, and a measure of efficiency, combining cost and time, needs to be developed in order to clearly establish where the optimum performance lies.

It is still unclear what limits there are to the approach, though the empirical results suggest that there is some upper limit for this coding and evaluation scheme, for which only a certain number of rows are actively deciding the cover.

It has been observed that mutation becomes a main thriving force once the algorithm approaches near-optimum solutions. There is still the question of how well this observation scales to bigger problems. The idea is that in the test problems used, the amount of search space left to mutation is not too big and may be time-efficiently covered, but for sizable problems this might not be true.

The number off different combinations in the parameters that the program accepts has made it impossible to try every single combination out. The test plan attempted to establish good parameter values by decoupling the individual effects from each other. Some insight needs to be gained into how the parameters interact with each other. Another important element to be considered is how dependent the optimal parameter values are on the problem. Only the initialisation tournament has been clearly shown to be problem dependent.

All in all, this seems a viable approach, worth further investigation.

7.2 What else?

As pointed out in the last section, there are a number of questions still open that should be addressed in future work. Most of them come down to widening the test problem set to cover bigger and more varied problems. This work pointed out some important directions and parameters that should be further investigated in more comprehensive research.

Some factors to be addressed would be:

- To determine whether it is true that the approach can be applied to other codings and/or evaluation schemes. This would require, for example, implementing other known genetic algorithms for the set covering problem like that reported in [Beasley & Chu 95]. The main objective here would be to test against their results in terms of solution quality and time efficiency.

- This leads to the next point which is the development of a combined cost-versus-time efficiency measure to search for the optimum pool sizes. A meta-level genetic algorithm could then be used to optimise these as well as other parameters, like crossover type and mutation rate.

- As pointed out in section 3.3.4, a possible evaluation scheme that might speed things up would allow pools to evolve independently for some generations, interleaving that with the combined fitness assignment, at least at an initial stage.

- Lastly, it seems interesting to test if the approach can be used to generate a highly fit population that can then be used by a traditional genetic algorithm to produce higher quality solutions in less time than it would otherwise require.

As a longer-term goal, our approach also needs to be tested on different though related problems, to try to extract problem-independent characteristics. The set partitioning problem seems to be an ideal candidate towards this goal.

Bibliography

[Balas & Ho 80] E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. *Mathematical Programming Study*, 12:37–60, 1980.

[Beasley & Chu 95] J.E. Beasley and P.C. Chu. A genetic algorithm for the set covering problem. Submitted to the European Journal of Operational Reseach, June 1995.

[Beasley 90] J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[Beasley *et al.* 93] D. Beasley, Bull D. R., and R. R. Martin. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[Brindle 81] A. Brindle. *Genetic Algorithms for Function Optimization.* Unpublished PhD thesis, University of Alberta, Edmonton, 1981.

[David & Steenstrup 87] L. David and M. Steenstrup. Genetic algorithms and simulated annealing: An overview. In L. Davis, editor, *Genetic Algorithm and Simulated Annealing*, pages 1–11. Morgan Kaufmann, Los Altos, California, U.S.A., 1987.

[Davidor 91] Y. Davidor. A naturally occuring niche and species phenomenon: the model and first results. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263. Morgan Kaufmann, 1991.

[DeJong 75] K. DeJong. *The Analysis and behaviour of a Class of Genetic Adaptive Systems.* Unpublished PhD thesis, University of Michigan, 1975.

[DeJong 85] K. DeJong. Genetic algorithms: A 10 year perspective. In J. J. Greffenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 169–177. Lawrence Erlbaum Associates, 1985.

[Eshelman *et al.* 89] L. J. Eshelman, R. Caruna, and J.D. Schaffer. Biases in the crossover landscape. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19. Morgan Kaufman, 1989.

[Fisher & Kedia 90] M.L. Fisher and P. Kedia. Optimal solution of set covering/partitioning problems using dual heuristics. *Management Science*, 36:674–688, 1990.

[Garey & Johnson 79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[Goldberg 89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA., 1989.

[Holland 75] J.H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[Jacobs & Brusco 93] L.W. Jacobs and M.J. Brusco. A simulated annealing-based heuristic for the set-covering problem. Working paper, Operations Management and Information Systems Department, Northern Illinois University, Dekalb,IL 60115, USA, March 1993.

[Liepins & Hilliard 87] G.E. Liepins and M. R. Hilliard. Greedy genetics. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithm on Genetic Algorithms*, pages 90–99. Lawrence Erlbaum Associates, 1987.

[Liepins & Potter 91] G. E. Liepins and W. D. Potter. A genetic algorithm approach to multiple-fault diagnosis. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 17, pages 237–250. Van Nostrand Reinhold, New York, 1991.

[Makins 91] M. Makins, editor. *English Dictionary.* Harper Collins Publishers, 1991.

[Marsten & Shepardson 81] R.F. Marsten and F. Shepardson. Exact solution of crew scheduling problems using the set partitioning model: Recent successful applications. *Networks*, 11(2):165–177, 1981.

[Michalewicz 94] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer Verlag, second edition, 1994.

[Nissen 93] V. Nissen. Evolutionary algorithms in management science: An overview and list of references. Number 9303 in Papers on Economics and Evolution. European Study Group for Evolutionary Economics, Interdisziplinares Graduiertenkolleg, Universitaet Goettingen, Gosslerstr. 12a, D-37073 Goettingen, Germany, July 1993.

[Schaffer & Eshelman 91] J. D. Schaffer and L. J. Eshelman. On crossover as an evolutionary viable strategy. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufman, 1991.

[Syswerda 89] G. Syswerda. Uniform crossover in genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989.

[Whitley & Starkweather 90] D. Whitley and T. Starkweather. Genitor ii: a distributed genetic algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, 2:189–214, 1990.

[Whitley 89] D. Whitley. The GENITOR algorithm and selective pressure. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.

[Whitley 94] D. Whitley. A genetic algorithm tutorial. In Z. Michalewicz, editor, *Special Issue on Evolutionary Compuation*, volume 4 of *Statistics and Computing*, pages 65–85. 1994.

Appendix A

Create Program

In order to generate random set covering problems, allowing the tester to set the column and row number, a small program was developed. The code is reproduced below. The inputs used are the size of the problem and the percentage of the density. The costs are randomly allocated, according to a uniform distribution over the interval $[0; columns/\_CostProportion]$.

```
/*

*******************************************************************************
**                                                                           **
**          MSc Information Technology: Knowledge Based Systems              **
**                 Divide and Conquer Genetic Algorithms                     **
**                     for the Set-Covering Problem                          **
**                                                                           **
**   Module:     Problem Generation                                          **
**   Author:     Luis F. Gonzalez Hernandez                                  **
**   Created:    Thu Aug 17 12:00:00 1995                                    **
**   Last Modified: Thu Aug 17 14:22:58 1995                                 **
**   Purpose:    Creates Random Set Covering Problems                        **
**   Remarks:    The costs are assigned randomly following a uniform         **
**               distribution in the interval: 0 - columns/_CostProportion   **
**                                                                           **
*******************************************************************************

*/

#define _CostProportion (5)
```

```
#include <stdlib.h>
#include <stdio.h>

main(int argc,char **argv)
{
  int density=2;
  int rows=0,columns=0,tempint,nrow,ncol,val;
  int random;
  int **scp,*ncoverers;
  FILE *datafile;
  char datafilename[256];

  /* Initialisation */
  strcpy(datafilename,"scpdata");
  printf("Random Set Covering Problem Generator.\n");
  printf("Luis F. Gonzalez (c) August 1995.\n");

  /* Handling Command-line Arguments */
  if(argc>2)
    {
      fflush(stdout);
      fprintf(stderr,"\nIncorrect number of arguments.\n");
      exit(-1);
    }
  if(argc==2)
    switch(argv[1][0])
      {
      case '-':
        printf("\nUsage: create [datafilename]\n");
        exit(0);
        break;
      default:
        strcpy(datafilename,argv[1]);
      }

  /* Read the density in */
  printf("\nDensity: ");
  scanf("%d",&tempint);
  if(tempint>0)
    density=tempint;
  else
    if(tempint<0)
      {
        fprintf(stderr,"\nCREATE: Invalid density value %d\n",tempint);
        exit(-1);
      }
```

```
/* Read the size of the problem */
/* Rows */
printf("Rows: ");
scanf("%d",&tempint);
if(tempint<=0)
  {
    fprintf(stderr,"\nCREATE: Invalid row number %f\n",tempint);
    exit(-1);
  }
rows=tempint;

/* Columns */
printf("Columns: ");
scanf("%d",&tempint);
if(tempint<=0)
  {
    fprintf(stderr,"\nCREATE: Invalid column number %f\n",tempint);
    exit(-1);
  }
columns=tempint;

/* Output log */
printf("\nThe parameters of the problem are:\n");
printf("Density: %d\tRows: %d\tColumns: %d\n",density,rows,columns);
printf("Creating the data, please be patient.\n");

/* Memory Allocation */
scp=(int **)malloc(rows*sizeof(int *));
ncoverers=(int *)malloc(rows*sizeof(int));
if((scp==(int **)NULL) ||
   (ncoverers==(int *)NULL))
  {
    fprintf(stderr,"\nCREATE: Unable to allocate memory.\n");
    exit(-1);
  }

/* Creating the Problem */
srand48((int)time(0));      /* Uses the time function to produce a seed */
for(nrow=0;nrow<rows;nrow++)
  {
    scp[nrow]=(int *)malloc(columns*sizeof(int));
    if(scp[nrow]==(int *)NULL)
      {
        fprintf(stderr,"\nCREATE: Unable to allocate memory.\n");
        exit(-1);
      }
    ncoverers[nrow]=0;
```

```
      for(ncol=0;ncol<columns;ncol++)
        {
          if((int)(drand48()*100)<density)
            {
              scp[nrow][ncol]=1;
              ncoverers[nrow]++;
            }
          else
            scp[nrow][ncol]=0;
        }
    }

  /* Creating the data file */
  datafile=fopen(datafilename,"w");
  if(datafile==(FILE *)NULL)
    {
      fprintf(stderr,"\nCREATE: Unable to create file %s.",datafilename);
      exit(-1);
    }
  fprintf(datafile,"%d ",rows);
  fprintf(datafile,"%d\n",columns);
  for(ncol=0;ncol<columns;ncol++)
    fprintf(datafile,"%d\n",
            ((int)(drand48()*(columns/_CostProportion))+1));
  for(nrow=0;nrow<rows;nrow++)
    {
      fprintf(datafile,"\n%d\n",ncoverers[nrow]);
      for(ncol=0;ncol<columns;ncol++)
        if(scp[nrow][ncol])
          fprintf(datafile,"%d ",ncol+1);
      fprintf(datafile,"\n");
    }
  fclose(datafile);
  free(ncoverers);
  for(nrow=0;nrow<rows;nrow++)
    free(scp[nrow]);
  free(scp);
}
```

Appendix B

Operations Research Library Problems: Optimal Covers

The problems used for testing have known optimal costs. Sample covers are provided below, as obtained by Paul C. H. Chu[1]

[1]Paul C. H. Chu can be contacted at the following address: Ph.D. Program, The Management School, Imperial College, London SW7 2AZ, England; or by electronic mail p.chu@ic.ac.uk

B.1 Problem 4.1

Cost	Number of Sets
429	65

0	1	2	4	5	7	8	9	10	11
12	13	14	15	16	17	20	21	22	24
25	27	28	42	43	45	46	47	48	49
51	53	57	58	61	62	65	68	69	70
74	76	77	80	84	85	88	90	93	102
106	115	119	120	121	123	128	137	143	145
152	168	193	274	432					

B.2 Problem 4.2

Cost	Number of Sets
512	60

0	1	2	4	5	7	9	10	14	16
17	18	19	21	22	24	28	31	40	43
46	47	48	50	51	53	55	58	59	60
61	62	63	68	69	70	75	78	86	87
91	93	100	102	111	112	113	119	135	136
144	159	160	162	167	176	208	273	293	375

B.3 Problem 4.3

Cost	Number of Sets
516	64

0	1	2	3	5	7	8	9	11	12
17	18	19	21	22	23	24	25	27	28
29	31	32	33	37	38	39	45	46	47
48	50	54	55	56	59	61	63	65	66
68	75	77	82	83	91	95	101	115	121
123	131	133	135	141	149	155	165	170	180
197	228	282	421						

Appendix C

Results obtained in the preliminary experiments

Pool	Assignment	Mutation	Mean	Std. Dev.
1	Random	Non-Adaptive	894.10	97.10
1	Random	Adaptive	796.61	97.40
1	Sequential	Non-Adaptive	798.10	63.33
1	Sequential	Adaptive	826.51	49.01

Table C–1: Experiments with one Pool

Pool	Assignment	Mutation	Mean	Std. Dev.
1	Random	Non-Adaptive	594.10	50.25
2	Random	Non-Adaptive	594.10	50.25
3	Random	Non-Adaptive	594.10	50.25
4	Random	Non-Adaptive	594.10	50.25
5	Random	Non-Adaptive	594.10	50.25
1	Random	Adaptive	609.00	32.46
2	Random	Adaptive	609.00	32.46
3	Random	Adaptive	609.00	32.46
4	Random	Adaptive	609.00	32.46
5	Random	Adaptive	609.00	32.46
1	Sequential	Non-Adaptive	629.45	48.03
2	Sequential	Non-Adaptive	629.84	48.12
3	Sequential	Non-Adaptive	630.04	48.18
4	Sequential	Non-Adaptive	628.35	47.95
5	Sequential	Non-Adaptive	628.70	47.95
1	Sequential	Adaptive	620.00	53.96
2	Sequential	Adaptive	620.00	53.96
3	Sequential	Adaptive	620.00	53.96
4	Sequential	Adaptive	620.00	53.96
5	Sequential	Adaptive	620.00	53.96

Table C–2: Experiments with five Pools

Pool	Mutation	Mean	Std. Dev.
1	Non-Adaptive	514.90	28.04
2	Non-Adaptive	514.90	28.04
3	Non-Adaptive	514.90	28.04
4	Non-Adaptive	514.90	28.04
5	Non-Adaptive	514.90	28.04
6	Non-Adaptive	514.90	28.04
7	Non-Adaptive	514.90	28.04
8	Non-Adaptive	514.90	28.04
9	Non-Adaptive	514.90	28.04
10	Non-Adaptive	514.90	28.04
11	Non-Adaptive	514.93	28.03
12	Non-Adaptive	514.90	28.04
13	Non-Adaptive	514.90	28.04
14	Non-Adaptive	514.92	28.03
15	Non-Adaptive	514.90	28.04
1	Adaptive	524.03	23.12
2	Adaptive	524.26	22.92
3	Adaptive	524.21	22.95
4	Adaptive	524.26	22.92
5	Adaptive	524.34	22.86
6	Adaptive	524.15	23.01
7	Adaptive	523.80	23.33
8	Adaptive	524.15	23.01
9	Adaptive	524.28	22.90
10	Adaptive	524.32	22.87
11	Adaptive	524.03	23.12
12	Adaptive	524.15	23.01
13	Adaptive	524.15	23.01
14	Adaptive	524.32	22.87
15	Adaptive	524.15	23.01

Table C–3: Experiments with fifteen Pools and Random Assignment

Pool	Mutation	Mean	Std. Dev.
1	Non-Adaptive	506.70	49.85
2	Non-Adaptive	506.70	49.85
3	Non-Adaptive	506.70	49.85
4	Non-Adaptive	506.70	49.85
5	Non-Adaptive	506.70	49.85
6	Non-Adaptive	506.70	49.85
7	Non-Adaptive	506.70	49.85
8	Non-Adaptive	506.70	49.85
9	Non-Adaptive	506.70	49.85
10	Non-Adaptive	506.70	49.85
11	Non-Adaptive	506.70	49.85
12	Non-Adaptive	506.70	49.85
13	Non-Adaptive	506.70	49.85
14	Non-Adaptive	506.70	49.85
15	Non-Adaptive	506.70	49.85
1	Adaptive	516.24	50.83
2	Adaptive	516.23	50.83
3	Adaptive	516.20	50.88
4	Adaptive	516.23	50.83
5	Adaptive	516.23	50.83
6	Adaptive	516.23	50.83
7	Adaptive	516.20	50.88
8	Adaptive	516.20	50.88
9	Adaptive	516.23	50.83
10	Adaptive	516.20	50.88
11	Adaptive	516.22	50.84
12	Adaptive	516.23	50.83
13	Adaptive	516.24	50.83
14	Adaptive	516.20	50.88
15	Adaptive	516.25	50.81

Table C–4: Experiments with fifteen Pools and Sequential Assignment

Pool	Mutation	Mean	Std. Dev.
1	Non-Adaptive	512.19	30.92
2	Non-Adaptive	512.53	30.49
3	Non-Adaptive	512.26	30.81
4	Non-Adaptive	512.57	30.94
5	Non-Adaptive	512.50	30.97
6	Non-Adaptive	512.39	30.98
7	Non-Adaptive	511.95	31.55
8	Non-Adaptive	511.91	31.38
9	Non-Adaptive	511.77	31.65
10	Non-Adaptive	512.11	31.34
11	Non-Adaptive	512.14	31.21
12	Non-Adaptive	511.83	31.52
13	Non-Adaptive	512.19	30.92
14	Non-Adaptive	512.04	31.17
15	Non-Adaptive	510.93	32.98
16	Non-Adaptive	512.50	30.52
17	Non-Adaptive	512.29	31.33
18	Non-Adaptive	512.21	30.93
19	Non-Adaptive	512.11	31.07
20	Non-Adaptive	511.31	33.24
21	Non-Adaptive	511.05	33.24
22	Non-Adaptive	511.46	32.31
23	Non-Adaptive	512.34	31.00
24	Non-Adaptive	512.16	31.16
25	Non-Adaptive	511.90	31.49

Table C–5: Experiments with twenty-five Pools and Random Assignment and Non-Adaptive Mutation

Pool	Mutation	Mean	Std. Dev.
1	Adaptive	526.93	34.71
2	Adaptive	526.94	34.74
3	Adaptive	526.89	34.74
4	Adaptive	526.95	34.79
5	Adaptive	527.00	34.77
6	Adaptive	526.98	34.81
7	Adaptive	526.95	34.74
8	Adaptive	526.93	34.75
9	Adaptive	526.90	34.74
10	Adaptive	526.98	34.77
11	Adaptive	526.97	34.75
12	Adaptive	526.94	34.76
13	Adaptive	526.95	34.69
14	Adaptive	526.91	34.74
15	Adaptive	526.95	34.73
16	Adaptive	526.90	34.70
17	Adaptive	526.89	34.79
18	Adaptive	526.85	34.69
19	Adaptive	526.95	34.79
20	Adaptive	526.96	34.76
21	Adaptive	526.94	34.74
22	Adaptive	526.92	34.74
23	Adaptive	526.92	34.80
24	Adaptive	526.93	34.73
25	Adaptive	526.92	34.74

Table C–6: Experiments with twenty-five Pools, Random Assignment and Adaptive Mutation

Pool	Mutation	Mean	Std. Dev.
1	Non-Adaptive	494.53	9.52
2	Non-Adaptive	494.52	9.53
3	Non-Adaptive	494.50	9.54
4	Non-Adaptive	494.52	9.53
5	Non-Adaptive	494.45	9.58
6	Non-Adaptive	494.52	9.53
7	Non-Adaptive	494.45	9.58
8	Non-Adaptive	494.50	9.54
9	Non-Adaptive	494.47	9.57
10	Non-Adaptive	494.50	9.54
11	Non-Adaptive	494.50	9.54
12	Non-Adaptive	494.50	9.54
13	Non-Adaptive	494.53	9.52
14	Non-Adaptive	494.45	9.58
15	Non-Adaptive	494.50	9.54
16	Non-Adaptive	494.50	9.54
17	Non-Adaptive	494.45	9.58
18	Non-Adaptive	494.48	9.55
19	Non-Adaptive	494.50	9.54
20	Non-Adaptive	494.47	9.56
21	Non-Adaptive	494.47	9.57
22	Non-Adaptive	494.52	9.53
23	Non-Adaptive	494.50	9.54
24	Non-Adaptive	494.50	9.54
25	Non-Adaptive	494.50	9.54

Table C–7: Experiments with twenty-five Pools, Sequential Assignment and Non-Adaptive Mutation

Pool	Mutation	Mean	Std. Dev.
1	Adaptive	512.60	33.78
2	Adaptive	512.75	34.07
3	Adaptive	512.80	34.17
4	Adaptive	512.60	33.78
5	Adaptive	512.75	34.07
6	Adaptive	512.75	34.07
7	Adaptive	512.70	33.97
8	Adaptive	512.68	33.92
9	Adaptive	512.78	34.13
10	Adaptive	512.75	34.07
11	Adaptive	512.78	34.13
12	Adaptive	512.75	34.07
13	Adaptive	512.68	33.92
14	Adaptive	512.72	34.01
15	Adaptive	512.75	34.07
16	Adaptive	512.75	34.07
17	Adaptive	512.75	34.07
18	Adaptive	512.80	34.17
19	Adaptive	512.60	33.78
20	Adaptive	512.75	34.07
21	Adaptive	512.70	33.97
22	Adaptive	512.72	34.01
23	Adaptive	512.60	33.78
24	Adaptive	512.75	34.07
25	Adaptive	512.75	34.07

Table C–8: Experiments with twenty-five Pools, Sequential Assignment and Adaptive Mutation

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
1	1274.52	84.63	1073.51	255.84
2	1272.36	85.67	1066.83	320.77
3	1273.15	88.04	1101.66	265.05
4	1273.55	88.90	1081.43	377.30
5	1273.58	83.96	1076.35	284.81
6	1273.68	85.74	1089.55	244.69
7	1274.38	84.78	1089.72	296.26
8	1275.05	88.00	1075.48	372.13
9	1272.38	89.12	1076.57	355.86
10	1272.82	89.34	1092.91	293.90
11	1271.20	88.57	1080.40	288.12
12	1275.48	86.11	1076.48	290.01
13	1271.88	88.52	1081.48	301.13
14	1271.16	87.76	1069.37	274.91
15	1271.01	88.16	1092.48	329.15
16	1272.81	85.32	1062.74	311.62
17	1274.96	88.56	1087.42	358.15
18	1271.84	85.75	1066.88	267.03
19	1273.15	87.64	1072.12	254.40
20	1273.23	87.02	1087.25	285.38
21	1277.39	85.91	1070.05	380.06
22	1272.31	82.08	1092.19	285.79
23	1267.60	87.14	1039.17	307.19
24	1271.57	83.49	1069.57	362.77
25	1270.26	85.61	1075.44	284.85
26	1270.73	85.38	1076.38	319.25
27	1272.75	88.52	1072.28	303.60
28	1273.27	87.44	1070.81	328.23
29	1270.84	86.61	1082.96	301.29
30	1270.60	88.22	1075.46	345.67
31	1272.87	86.79	1080.34	262.38
32	1268.96	89.50	1076.99	354.45
33	1267.05	87.44	1045.07	352.77
34	1270.24	87.40	1067.62	267.75
Continues				

Table C–9: Experiments with fifty Pools, Non-Adaptive Mutation and Random Assignment

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
35	1270.67	84.95	1050.50	272.91
36	1272.34	86.67	1104.43	272.70
37	1268.59	87.87	1065.44	291.87
38	1274.56	89.54	1095.17	368.75
39	1271.34	85.81	1070.39	363.96
40	1273.99	86.52	1106.47	275.33
41	1269.75	89.73	1070.03	363.40
42	1273.48	83.55	1070.14	233.97
43	1271.88	83.55	1080.10	256.53
44	1275.70	85.93	1079.75	300.30
45	1275.58	86.09	1073.40	214.34
46	1272.06	87.48	1062.74	427.12
47	1270.70	85.59	1061.20	255.24
48	1273.15	86.33	1084.15	302.76
49	1267.86	85.12	1064.55	280.97
50	1270.78	88.43	1066.61	360.62

Table C–10: Experiments with fifty Pools, Non-Adaptive Mutation and Random Assignment (Cont.)

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
1	1243.01	64.45	1054.40	192.29
2	1245.12	66.16	1038.80	197.47
3	1244.18	62.71	1047.75	207.11
4	1248.04	65.68	1062.99	228.16
5	1241.87	64.32	1044.54	217.75
6	1242.05	65.88	1043.31	241.19
7	1240.20	67.54	1037.65	221.55
8	1245.56	66.22	1064.70	253.19
9	1244.82	65.41	1054.65	278.24
10	1242.79	62.17	1045.40	197.52
11	1242.30	66.27	1066.09	219.12
12	1244.16	67.74	1059.98	225.75
13	1242.34	66.56	1056.43	220.62
14	1241.42	65.79	1061.41	251.46

Continues

Table C–11: Experiments with fifty Pools, Adaptive Mutation and Random Assignment

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
15	1242.87	63.50	1054.62	249.64
16	1243.16	64.37	1051.17	200.63
17	1242.31	64.09	1058.29	238.90
18	1243.80	62.88	1052.59	231.50
19	1245.35	65.74	1046.94	231.40
20	1242.73	63.22	1040.78	234.04
21	1244.64	63.98	1032.71	268.70
22	1243.03	63.64	1055.34	215.74
23	1242.49	61.08	1050.13	219.62
24	1243.57	65.66	1048.85	266.59
25	1243.26	63.27	1039.66	235.35
26	1242.03	61.63	1041.06	183.73
27	1244.58	63.43	1038.01	188.80
28	1245.85	63.97	1052.91	176.14
29	1245.18	64.44	1050.49	241.06
30	1240.40	67.25	1050.19	264.65
31	1241.42	64.38	1033.26	218.86
32	1241.56	65.91	1055.38	239.92
33	1245.42	60.32	1068.20	222.24
34	1246.52	64.80	1041.78	211.32
35	1240.81	65.92	1042.71	268.45
36	1244.92	64.37	1077.01	177.84
37	1241.70	61.40	1009.64	240.50
38	1242.71	63.21	1065.29	225.72
39	1243.78	62.90	1056.13	218.35
40	1246.61	65.03	1030.30	235.57
41	1245.39	62.64	1061.12	172.79
42	1244.50	63.70	1062.77	226.48
43	1239.39	67.81	1037.88	208.38
44	1245.92	63.87	1055.61	286.05
45	1245.09	64.24	1054.80	201.83
46	1242.15	68.63	1044.26	233.76
47	1242.27	62.86	1092.01	172.24
48	1243.85	67.25	1030.11	254.27
49	1241.84	62.77	1068.81	227.32
50	1241.98	64.60	1049.61	221.64

Table C–12: Experiments with fifty Pools, Adaptive Mutation and Random Assignment (Cont.)

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
1	1190.64	105.22	1011.54	317.64
2	1187.73	106.97	1017.71	301.45
3	1182.86	109.67	972.04	397.44
4	1188.39	107.15	990.89	346.87
5	1185.88	107.45	1011.48	335.43
6	1188.85	108.98	1018.95	318.66
7	1188.04	106.16	1002.47	280.99
8	1189.44	107.95	1016.20	329.36
9	1186.63	110.34	1015.28	380.38
10	1186.71	108.70	991.87	341.51
11	1188.97	107.81	1027.52	324.33
12	1187.26	109.08	1012.83	258.82
13	1187.54	111.19	991.24	312.24
14	1186.48	105.87	1000.58	358.41
15	1190.58	106.94	1006.84	326.07
16	1186.84	104.22	999.83	228.50
17	1189.10	101.35	998.33	315.90
18	1184.96	103.97	1007.98	257.37
19	1187.89	109.12	997.98	284.59
20	1186.51	108.48	998.80	343.38
21	1189.63	109.00	1014.49	350.42
22	1188.77	106.56	998.72	289.30
23	1185.65	109.10	990.02	342.11
24	1189.29	108.47	1016.55	353.14
25	1187.91	106.39	996.84	301.17
26	1186.84	109.37	987.31	300.15
27	1186.22	109.21	1004.56	330.46
28	1189.16	107.21	993.34	311.25
29	1183.80	111.56	1023.57	394.11
30	1185.48	108.21	990.17	282.82
31	1187.81	107.04	1001.06	301.83
32	1188.36	109.46	998.53	319.92
33	1186.53	107.96	999.69	302.54
34	1183.91	106.48	994.95	294.29

Continues

Table C–13: Experiments with fifty Pools, Non-Adaptive Mutation and Sequential Assignment

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
35	1186.59	107.63	1000.39	275.28
36	1189.00	105.93	1007.00	316.91
37	1184.27	104.84	978.19	293.06
38	1188.00	108.79	1025.59	343.42
39	1188.02	107.72	1008.94	305.59
40	1186.68	108.63	998.55	358.72
41	1184.67	105.38	992.33	365.42
42	1190.96	105.18	1012.44	281.99
43	1189.49	108.27	1020.06	274.90
44	1184.69	109.02	1002.33	379.35
45	1184.32	106.37	999.94	249.62
46	1188.50	109.45	980.14	312.16
47	1185.66	105.80	991.61	290.79
48	1187.15	106.64	1013.77	300.63
49	1188.49	107.51	1014.23	356.33
50	1189.27	103.77	1030.10	297.65

Table C–14: Experiments with fifty Pools, Non-Adaptive Mutation and Sequential Assignment (Cont.)

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
1	1167.28	96.01	971.53	306.00
2	1168.70	97.31	976.38	306.37
3	1168.03	95.47	997.32	237.27
4	1169.08	96.51	985.42	251.06
5	1167.66	95.59	990.59	335.86
6	1167.09	98.45	977.03	298.00
7	1164.83	95.86	970.69	299.55
8	1168.94	98.02	980.52	316.45
9	1164.56	95.25	967.01	376.38
10	1166.75	97.11	991.25	236.81
11	1167.52	95.24	1004.86	258.32
12	1169.56	97.46	976.66	320.31
13	1166.32	99.44	969.45	291.93
14	1166.35	95.15	989.07	285.02
Continues				

Table C–15: Experiments with fifty Pools, Adaptive Mutation and Sequential Assignment

Pool	Mean - Average Score	Std. Dev. - Average Score	Mean	Std. Dev.
15	1165.18	96.42	989.07	296.98
16	1166.05	96.95	979.56	276.30
17	1164.37	95.15	966.74	287.68
18	1164.28	95.24	967.06	223.73
19	1165.33	97.14	980.10	260.99
20	1167.92	97.82	997.13	318.19
21	1167.86	97.82	983.57	335.11
22	1169.71	96.47	985.82	319.44
23	1167.47	92.99	978.50	230.40
24	1167.33	96.62	980.15	331.20
25	1166.26	96.14	978.60	321.16
26	1168.59	95.29	991.94	269.18
27	1167.48	98.10	968.11	306.89
28	1166.31	97.49	1005.21	286.71
29	1168.86	94.90	999.13	297.89
30	1165.46	96.92	984.67	283.38
31	1166.51	95.69	945.21	333.95
32	1167.88	97.09	998.00	290.37
33	1169.06	94.00	999.67	281.72
34	1166.19	97.54	984.88	274.13
35	1162.90	93.65	961.82	233.75
36	1167.47	95.59	974.88	290.16
37	1167.04	93.37	976.32	260.98
38	1166.22	97.81	978.97	319.94
39	1169.13	96.00	979.74	280.78
40	1165.85	100.81	984.02	361.38
41	1167.16	94.24	988.44	278.50
42	1169.03	97.88	999.64	294.73
43	1167.02	95.82	986.60	267.41
44	1169.87	97.61	995.81	336.62
45	1165.07	98.44	977.80	298.88
46	1168.75	97.96	973.65	278.09
47	1165.15	96.23	985.22	283.27
48	1166.97	98.02	985.13	321.32
49	1166.51	101.09	971.06	376.45
50	1166.50	94.75	990.23	316.42

Table C–16: Experiments with fifty Pools, Adaptive Mutation and Sequential Assignment (Cont.)

Appendix D

DCSCPGA Code

```
/*

******************************************************************************
**                                                                          **
**            MSc Information Technology: Knowledge Based Systems           **
**                  Divide and Conquer Genetic Algorithms                   **
**                      for the Set-Covering Problem                        **
**                                                                          **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Last Modified: Thu Aug 10 13:57:58 1995                                **
**                                                                          **
******************************************************************************

*/

/*
   Structures used to hold the set covering problem data read from the data
   file.
*/

typedef struct scp_row {
  int ncoverers;  /* Number of sets that cover that specific element/row */
  int *coverers;  /* Pointer to a matrix of the individual subsets that
                     cover that element/row */
} SCP_ROW;

typedef struct dc_row {
/*  int subproblem;  Subproblem this element is member of */
  int size;       /* Size of the subproblem -Sequential Assignment-      */
  int *rows;      /* Rows covered in this subproblem -Random Assignment- */
  SCP_ROW *dc_row_coverage; /* Pointer to row coverage information */
} DC_ROW;

typedef struct scp_col {
/*  int subproblem; Subproblem this structure is refering to */
  int cost;       /* Cost associated with the specific column */
  int ncovers;    /* Number of elements this set covers in this specific
                     pool */
  int *covers;    /* Pointer to a matrix of the individual elements that
```

```
                    this set covers in this subproblem */
} SCP_COL;


/*
   Structures used to hold the genetic data
*/

typedef struct genodata {     /* genotype string plus reference count    */
  int *genotype;
  int refcount;
  int id;
  int parent1, parent2;
} GENODATA;

typedef struct chromosome {   /* A chromosome consists of a genotype and */
  GENODATA *gdp;               /* a fitness.                              */
  double fitness;
} CHROMOSOME;

typedef struct chromojoint{   /* Several CHROMOSOMES that together are   */
  int *whatchunks;            /**** FOR CREDIT ASSIGNMENT ****/
  CHROMOSOME *chunks;         /* a solution to the problem and each      */
  int ncover, *cover;         /* belonging to a subproblem pool. */
  int nactive, *active;
} CHROMOJOINT;

typedef struct chromopair {   /* Returned by crossover, contains one or */
  CHROMOSOME child1;          /* two children depending on -t flag      */
  CHROMOSOME child2;
} CHROMOPAIR;

typedef struct subproblem_pool { /* Used to refer to the individual pools */
  int nmembers;   /* Number of CHROMOSOMEs present in this population */
  CHROMOSOME *population; /* Pointer to the population itself */
} SUBPROBLEM_POOL;



/*
   Global Variables
*/
SUBPROBLEM_POOL *DC_solution; /* Pointer to a dynamically created matrix that
                               will hold all the populations currently in
                               use. */
SCP_COL *DC_col_coverage; /* Pointer to each of the SCP_COL_POOL structures
                               that will carry the information about how
                               which columns are covering which subproblem */
DC_ROW *DC_subproblem;

struct problem_data {
  int scp_rows;       /* Number of rows/elements in the SCP */
  int scp_cols;       /* Number of columns/sets in the SCP */
  int *scp_costs;     /* Pointer to a dynamically created matrix to hold
                         the costs of each subset. */
  int DC_nsubproblems; /* Number of subproblems that the systems is using */
} DC_problem;


#define FALSE (0)
```

```
#define TRUE  (1)

#ifndef _ASSIGNMENTS_
#define _ASSIGNMENTS_

#define _DC_RandomAssignment      0
#define _DC_SequentialAssignment  1

#endif

#ifndef _EVALUATIONS_
#define _EVALUATIONS_

#define _DC_AverageEvaluation       0
#define _DC_RandomEvaluation        1
#define _AVERAGE_EVALUATIONS (2)
#define _MIN_EVALUATIONS_THRESHOLD (2)

#endif

#ifndef _SELECTION_
#define _SELECTION_

#define _DC_ScoredSelection         0
#define _DC_RandomSelection         1

#endif


#ifndef _REPRODUCTION_
#define _REPRODUCTION_

#define _Offspring_Chunk (-1)

#endif

#define _NUMBER_CHROMOSOMES_IN_POPULATION (50)

/* User actions possible */

#define QUIT      1
#define CONTINUE  2
#define RESTART   3
#define ASK_AGAIN 4

/* data input files */

#define WEIGHTSFILE  "weights"  /* name of file for knapsack data */
#define RRDATAFILE   "rrdata"   /* name of file for Royal Road data */
#define TTDATAFILE   "ttdata"   /* Name of file for timetable data */
#define SCPDATAFILE  "scpdata"  /* name of file for set covering data */
#define GSPDATAFILE  "gspdata"  /* name of file for gene switch data */

/* On some operating systems you will need a different random (etc) */
#if hpux || SOLARIS
#ifdef SOLARIS
#include <stdlib.h>
#endif
#define drandom() drand48()
#define random() lrand48()
```

```
#define srandom(seed) srand48((long) seed)
#endif hpux || SOLARIS

#ifdef _DEFINE_VARS_
int numchromo=_NUMBER_CHROMOSOMES_IN_POPULATION;
int crowding_factor=5;
int scp_mutation_tournament;
int scp_init_mutation_tournament=1;
int numpop = 1,           /* Number of populations */
    gens_stage = 1000,     /* How many times to cycle before asking user */
    repinterval = 50,     /* Interval between screen/file reports */
    isadaptive = FALSE,   /* Using adaptive mutation? */
    last_line = 24,       /* Size of screen; actually found at run time */
    max_allele,           /* In most problems, alleles in range 0..this */
    tournament_size = 3,  /* Default size in tournament selection */
    chromo_id = 1;        /* Chromosome IDs start from this value */
int randseed;             /* The random seed */
FILE *output = (FILE *)NULL;       /* Output file if any */
FILE *chromofile = (FILE *)NULL;   /* Chromo file, if any output file */
FILE *coverfile = (FILE *)NULL;    /* Best Cover Sofar file */
int interactive = TRUE;            /* Interactive? */
int finaldump = FALSE;             /* Noninteractive: do final chromodump? */
int chromoutput = FALSE;           /* Any chromosome output? */
/*int twins = FALSE;*/      /* Crossover produces pairs, or just one child? */
char *datafilename = (char *)NULL;
char outfilename[256], chromofilename[256],coverfilename[256];
char prompt[80];
char *geno_size_string;    /* String naming size of a chromo (hack) */
double mute_rate = 0.02,   /* Default mutation rate */
       cross_rate = 0.6,   /* Default crossover rate if applicable */
       bias = 1.5;         /* Default bias for rank selection */

char *eval_name,           /* String giving name of problem */
     *maxfitness,          /* String giving max fitness (if known) */
     *cross_name,          /* String naming crossover type */
     *select_name,         /* String naming selection type */
     *repro_name,          /* String naming reproduction type */
     *mutate_name;         /* String naming mutation type */
double *total_abs_fitness; /* Used in roulette selection: size of wheel */
int evals;                 /* How often eval has been called */
double user_threshold;     /* Non-interactive? Stop when one/all reach this */
int n_for_stop;            /* How many must reach user_threshold to stop */
int assignment_type=_DC_RandomAssignment;
int selection_type=_DC_RandomSelection; /* Way in which other chunks are
                                           selected when evaluating one. */
int geno_size;
#else
extern int numchromo;
extern int crowding_factor;
extern int scp_mutation_tournament;
extern int scp_init_mutation_tournament;
extern char isbdsp;
extern int selection_type;
extern int geno_size;
extern int evals;
extern char *coverfilename;
extern char *datafilename;
#endif
```

```
/**********************************************************************
 *                                                                    *
 *              PGA - Parallel Genetic Algorithm Testbed              *
 *                                                                    *
 *          "A Parallel Implementation of a genetic algorithm"        *
 *                                                                    *
 *               By Peter Ross (peter@ed.ac.uk)                       *
 *                  Geoffrey H. Ballinger (geoff@ed.ac.uk),           *
 *                                                                    *
 **********************************************************************/

/*
   Copyright (C) 1993, 1994 Peter Ross and Geoffrey H. Ballinger.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
*/

#define TITLE    "Divide and Conquer Genetic Algorithm"
#define VERSION  "1.0"
#define AUTHOR   "Luis F. Gonzalez"
#define PGA      "SCPGA Dave Corne, Peter Ross, Geoff Ballinger (c) 1993,1994"
#define DATE     "July 1995"
```

```
/* screen.h
 * Copyright (C) 1993 Peter Ross and Geoff Ballinger.
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Locations for use in PGA screen layout.
 */

/* Version      */
#define VER_ROW       2
#define VER_COL       10

/* Populations: */
#define POP_ROW       3
#define POP_COL       1

/* Chromosomes per population: */
#define CPP_ROW       3
#define CPP_COL       35

/* ....Chromosome length: */
#define CRL_ROW       3
#define CRL_COL       39

/* Number of evaluations: */
#define EVLIM_ROW     4
#define EVLIM_COL     1

/* ...Reporting interval: */
#define REPINT_ROW    5
#define REPINT_COL    1

/* ...Migration interval: */
#define MIG_ROW       6
#define MIG_COL       1

/* .Reproduction: */
#define REPRO_ROW     4
#define REPRO_COL     43

/* ...Crossover type: */
#define CTYPE_ROW     5
#define CTYPE_COL     39

/* ...Crossover rate: */
#define CROSS_ROW     6
#define CROSS_COL     39

/* ....Mutation rate: */
#define MUT_ROW       7
#define MUT_COL       39

/* ....Log file name: */
#define FNAME_ROW     8
#define FNAME_COL     39

/* Eval function: */
#define EVAL_ROW      7
#define EVAL_COL      9
```

```
/* ....Selection: */
#define SELECT_ROW    8
#define SELECT_COL    9

/* ...Selection bias: */
#define BIAS_ROW      9
#define BIAS_COL      5

/* Generation: */
#define GEN_ROW       9
#define GEN_COL       46

/* Evaluations so far: */
#define EVCOUNT_ROW   10
#define EVCOUNT_COL   38

/* The population table */
#define POPTABLE_ROW  11
#define POPTABLE_COL  14
```

```
/*

******************************************************************************
**                                                                          **
**           MSc Information Technology: Knowledge Based Systems           **
**                 Divide and Conquer Genetic Algorithms                    **
**                     for the Set-Covering Problem                         **
**   Module:                                                                **
**   Routines:                                                              **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    Wed Jun 14 20:00:00 1995                                   **
**   Last Modified: Wed Aug  9 18:10:29 1995                                **
**                                                                          **
**   Routines in other modules:                                             **
**   Global Data Structures:                                                **
**                                                                          **
******************************************************************************

*/

#ifndef _ERROR_MESSAGES_
#define _ERROR_MESSAGES_

#define _OffsetHandle 100
#define _DC_HANDLE_POOL_CODE (1+_OffsetHandle)
#define _DC_HANDLE_POOL "DC_GA: Minimum subproblem number is one.\n"
#define _DC_HANDLE_NUMCHROMO_CODE (2+_OffsetHandle)
#define _DC_HANDLE_NUMCHROMO "DC_GA: Minimum chromosome number is one.\n"
#define _DC_HANDLE_INTERACTIVE_CODE (3+_OffsetHandle)
#define _DC_HANDLE_INTERACTIVE "DC_GA_non-interactive: specify -NOthreshold or -NAthreshold\n"
#define _DC_HANDLE_INTERACTIVE_TYPE "DC_GA_non-interactive: must say O (one) or A (all)\n"
#define _DC_HANDLE_NONINTERACTIVE_THRESHOLD "DC_GA_non-interactive: must set stop threshold\n"
#define _DC_HANDLE_REPORT_CODE (4+_OffsetHandle)
#define _DC_HANDLE_REPORT "DC_GA: Can't report every %d generations.\n"
#define _DC_HANDLE_SEED "DC_GA: Invalid Random seed. Using default.\n"
#define _DC_HANDLE_DATAFILE_CODE (5+_OffsetHandle)
#define _DC_HANDLE_DATAFILE "DC_GA: Use -Fdatafile (No spaces).\n"
#define _DC_HANDLE_ASSINGMENT_CODE (6+_OffsetHandle)
#define _DC_HANDLE_ASSINGMENT "DC_GA: I don't know about %s assignment.\n"
#define _DC_HANDLE_MUTATION_CODE (7+_OffsetHandle)
#define _DC_HANDLE_MUTATION "DC_GA: I don't know about %s mutation.\n"

#define _DC_HANDLE_MUTERATE_CODE (8+_OffsetHandle)
#define _DC_HANDLE_MUTERATE "DC_GA: Mutation rate must be within [0,1].\n"
#define _DC_HANDLE_CROSSRATE_CODE (9+_OffsetHandle)
#define _DC_HANDLE_CROSSRATE "DC_GA: Crossover rate must be within [0,1].\n"
#define _DC_HANDLE_BIAS_CODE (10+_OffsetHandle)
#define _DC_HANDLE_BIAS "DC_GA: Bias must be within (1,2).\n"
#define _DC_HANDLE_INITTOUR_CODE (11+_OffsetHandle)
#define _DC_HANDLE_INITTOUR "DC_GA: Initial mutation_tournament must be greater than one.\n"
#define _DC_HANDLE_SELECTION_CODE (12+_OffsetHandle)
#define _DC_HANDLE_TMARG "DC_GA_tm_selection Invalid size. Set to 10\n"
#define _DC_HANDLE_TNARG "DC_GA_tn selection Invalid size. Set to 10\n"
#define _DC_HANDLE_SELECTION "DC_GA: I don't know about %s selection.\n"
#define _DC_HANDLE_REPROD_CODE (13+_OffsetHandle)
#define _DC_HANDLE_REPROD "DC_GA: I don't know about %s reproduction.\n"
#define _DC_HANDLE_EVAL_MUTATE_CODE (14+_OffsetHandle)
#define _DC_HANDLE_EVAL_MUTATE "DC_GA: Not valid evaluation function.\n"
#define _DC_HANDLE_INVALIDARG  "Unknown argument %s.\n\n"
#define _DC_HANDLE_OUTFILE_CODE (15+_OffsetHandle)
```

```
#define _DC_HANDLE_OUTFILE "DC_GA: Unable to open %s for output.\n"


#define _OffsetReadFile 200

#define _SCP_FILE_OPEN_CODE (1 + _OffsetReadFile)
#define _SCP_FILE_OPEN "\nDC_file_read Error: Unable to open the datafile.\n"
#define _SCP_FILE_MALLOC_CODE (2 + _OffsetReadFile)
#define _SCP_FILE_MALLOC "\nDC_file_read Error: Unable to allocate Memory.\n"


#define _OffsetCredit 300

#define _DC_SELECT_MALLOC_CODE (1 + _OffsetCredit)
#define _DC_SELECT_MALLOC "\nDC_Selection Error: Unable to allocate Memory.\n"


#define _OffsetInit 400

#define _DC_INIT_MALLOC_CODE (1+_OffsetInit)
#define _DC_INIT_MALLOC "\nDC_Init Error: Unable to allocate Memory.\n"

#define _OffsetReprod 400
#define _DC_REPROD_MALLOC_CODE (1+_OffsetReprod)
#define _DC_REPROD_MALLOC "\nDC_Reprod Error: Unable to allocate Memory.\n"
#endif
```

```
/*

*****************************************************************************
**                                                                         **
**           MSc Information Technology: Knowledge Based Systems           **
**                 Divide and Conquer Genetic Algorithms                   **
**                     for the Set-Covering Problem                        **
**                                                                         **
**   Prototype Header File                                                 **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    Mon Jun 19 12:00:00 1995                                  **
**   Last Modified: Wed Aug 16 20:43:06 1995                               **
**                                                                         **
**                                                                         **
*****************************************************************************

*/

#ifndef _Prototypes
#define _Prototypes
int dc_create_subproblems(int, int *); /* Creates the datastructures that
                                           will hold the D&C information     */
void dc_fixedsize_pools(int, int, int *);    /* Sizes of the pools are fixed */
void dc_variablesize_pools(int, int, int *); /* Sizes of the pools are
                                                variable                     */

int dc_assign_random_values(int *); /* Does the random assignment of
                                        the elements to the pools            */
int dc_sort_compare(const void *,const void *); /* Compare function used for
                                                   qsort in the random
                                                   assingment                */


/*
   Module: Initialisation
*/
void init_pops_sorted(int);
void init_genotype(int, GENODATA *);
int compchromo(void const *,void const *);



/****************************************************************************
**                                                                         **
**   Module:     Credit Assignment                                         **
**   Routines:   dc_random_selection: selects a chromojoint randomly       **
**               dc_scored_selection: selects a chromojoint according to the **
**                                    fitness of its members               **
**               dc_getgenotype: returns the genotype in a chromosome      **
**               dc_getcoverage: returns the cover corresponding to some   **
**                               gene                                      **
**               dc_indextorow:  returns the row in some given position    **
**               dc_eval_scp: evaluates a complete cover                   **
**               scorecover:  assigns a score to a cover compared to the   **
**                            best sofar                                   **
**               intcompare:  compares two integers for sorting            **
**               dc_random_peer_selection: selects the rest of the chunks  **
**                                    randomly.                            **
**               dc_scored peer_selection: selects the rest of the chunks  **
**                                    according to their fitness.          **
```

```
**              dc_evaluate_solution: evaluates the current solution        **
**                                    population                            **
**                                                                          **
****************************************************************************/
CHROMOJOINT dc_random_selection();
CHROMOJOINT dc_scored_selection();
static int intcompare(const void *, const void *);
double dc_eval_scp(CHROMOJOINT *);
int dc_getcoverage(int,int);
int dc_getgenotype(CHROMOJOINT,int);
int dc_indextorow(int);
CHROMOJOINT dc_scored_peer_selection(int,int,CHROMOSOME);
CHROMOJOINT dc_random_peer_selection(int,int,CHROMOSOME);
void dc_evaluate_solution(int);
double scorecover(int *,int,int *,int, int *,int *);

/*
   Module: Reproduction
*/
void one_reproduction(void);
void one_cross_reproduction(void);
void insert(CHROMOJOINT,double *);
void gen_reproduction(void);

/*
   Functions developped by Peter Ross (c) 1993, 1994
   Slightly changed to allow their communication with
   the Selection and Credit Assignment module.
*/
CHROMOSOME rank_select(CHROMOSOME *,int *,int, int);
CHROMOSOME fitprop_select(CHROMOSOME *,int *,int, int);
CHROMOSOME tm_select(CHROMOSOME *,int *,int, int);
CHROMOSOME tn_select(CHROMOSOME *,int *,int, int);

CHROMOSOME mutate_one_bit(CHROMOSOME, double, int);
CHROMOSOME mutate_01(CHROMOSOME, double, int);
CHROMOSOME mutate_multi(CHROMOSOME, double, int);
double adaptive(CHROMOSOME,CHROMOSOME,int);
double nonadaptive(CHROMOSOME, CHROMOSOME,int);

CHROMOSOME one_pt_cross(CHROMOSOME, CHROMOSOME, int);
CHROMOSOME two_pt_cross(CHROMOSOME, CHROMOSOME, int);
CHROMOSOME uniform_cross(CHROMOSOME, CHROMOSOME, int);
CHROMOSOME no_cross(CHROMOSOME, CHROMOSOME, int);
int rand_bit();

void dcscpga_help(void);

int stringcat(char *,char *,int);
void setprompt(int,int,int);
int report(int,int,double);
void free_pops(void);
void handle(char *);
void dump_chromos(void);
void getcover(CHROMOSOME *, int *, int *);

CHROMOSOME (*mutate)(CHROMOSOME, double,int);
#endif
```

```
/*

*****************************************************************************
**                                                                         **
**          MSc Information Technology: Knowledge Based Systems           **
**                Divide and Conquer Genetic Algorithms                    **
**                     for the Set-Covering Problem                        **
**                                                                         **
**   Module:     Main                                                      **
**   Authors:    Peter Ross and Geoff Ballinger (c) 1993, 1994             **
**   Adapted by: Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Last Modified: Wed Sep 13 13:13:05 1995                               **
**                                                                         **
*****************************************************************************

*/
#include <stdio.h>            /* Include the IO and maths header files. */
#include <string.h>
#include <math.h>
#include <curses.h>

#define _DEFINE_VARS_

#include "dcscpga.h"  /* defines genodata and chromosome structs */
#include "prototypes.h"
#include "errormsg.h"
#include "version.h"             /* title string and version number */
#include "screen.h"              /* screen layout configuration */


void (*initialise)() = init_pops_sorted;
CHROMOSOME (*crossover)() = two_pt_cross;
int mutatemulti=FALSE;
void (*reproduction)() = one_cross_reproduction;
double (*mrate)(CHROMOSOME,CHROMOSOME,int) = nonadaptive;
CHROMOSOME (*selection)() = tn_select;
int scp=1;
char isbdsp=0;

/**********************************************************************/
/* This sets the prompt for next interaction                          */
/**********************************************************************/

int stringcat(char *s1,char *s2,int n)
{
  int i,j;
  for(i=n,j=0; s2[j] != '\0'; i++, j++)
      s1[i] = s2[j];
  s1[i] = '\0';
  return(i);
}

void setprompt(int filing,int saved,int satisfied)
{
  int i;
  for(i=0;i<80;i++) prompt[i]='\0';
  i=stringcat(prompt,"(A)gain, (Q)uit", 0);
  if(filing && !saved)
    i=stringcat(prompt, ", (S)ave", i);
```

```
  if(!satisfied)
    i=stringcat(prompt, ", (C)ontinue",i);
  stringcat(prompt, ": ", i);
}

/*******************************************************************/
/* This updates the screen and, if necessary, the output file          */
/* If interactive, returns the number that have converged.            */
/* If NOT interactive, returns the number that have passed the supplied */
/* threshold value.                                                   */
/*******************************************************************/

int report(int gen_total,int fileoutput,double threshold)
{
  int npool, chromo;
  int n_at_threshold = 0;
  char conv_char;
  double average, best, tmp;

  if(interactive)
    {
      mvprintw(GEN_ROW,GEN_COL+12,"%d",gen_total);         /* necessary */
      mvprintw(EVCOUNT_ROW,EVCOUNT_COL+20,"%d", evals);
    }
  if(fileoutput)
    {
      fprintf(output, "\nGenerations = %d  Evaluations-so-far = %d\n",
              gen_total,evals);
    }

  for(npool=0; npool<DC_problem.DC_nsubproblems;npool++)
    {
      average=0.0;
      best = -1000000.0;
      for(chromo=0; chromo<numchromo; chromo+=1)
        {
          tmp = DC_solution[npool].population[chromo].fitness;
          average+=tmp;
          if(best < tmp) best = tmp;
        }
      average/=(double)numchromo;
      if(fabs(average-best) < 1e-9)
        {
          conv_char = '=';
          n_at_threshold++;
        }
      else
        {
          conv_char = ' ';
          if(!interactive && best >= threshold)
            n_at_threshold++;
        }
      if(interactive)
        if(isbdsp)
          mvprintw(POPTABLE_ROW+1+npool,POPTABLE_COL,
                   "%-4d%c    %12.7f     %12.7f",
                   npool, conv_char,
                   (double)((10/average)+1.0),
                   (double)((10/best)+1.0));
        else
```

```
          mvprintw(POPTABLE_ROW+1+npool,POPTABLE_COL,
                   "%-4d%c    %12.7f     %12.7f",
                   npool, conv_char,
                   (double)((1/average)+1.0),
                   (double)((1/best)+1.0));
      if(fileoutput)
        if(isbdsp)
          fprintf(output, "  %-4d    %12.7f     %12.7f\n",
                  npool,    (double)((10/average)-1.0),
                  (double)((10/best)-1.0));
        else
          fprintf(output, "  %-4d    %12.7f     %12.7f\n",
                  npool,    (double)((1/average)-1.0),
                  (double)((1/best)-1.0));
    }
  if(interactive)
    refresh();
  return(n_at_threshold);
}

/**********************************************************************/
/* This frees whole population                                        */
/**********************************************************************/

void free_pops(void)
{
  CHROMOSOME *whole_pop;
  int npool,nchromo;

  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    {
      whole_pop=DC_solution[npool].population;
      for(nchromo=0;nchromo<DC_solution[npool].nmembers;nchromo++)
        {
          /* Avoid freeing anything twice */
          if(whole_pop[nchromo].gdp->refcount == 1)
            {
              free(whole_pop[nchromo].gdp->genotype);
              free(whole_pop[nchromo].gdp);
            }
          else
            whole_pop[nchromo].gdp->refcount--;
        }
      free(whole_pop);
    }
  free(DC_solution);
}


/**********************************************************************/
/* This resets the defaults according to the argument 'arg'.          */
/**********************************************************************/
void handle(char *arg)
{
  int newint, newint2;
  double newdouble;
  int nerror=0;
  char tt_eventchoice[16], tt_slotchoice[16];

  if (*arg=='-')
```

```
{               /* Either a switch ...        */
  switch (*(arg+1))
    {
    case 'P':  /* Sets the number of pools */
      if(sscanf(arg,"-P%d",&newint) == 1)
        {
          if(newint<1)
            {
              fprintf(stderr,_DC_HANDLE_POOL);
              exit(_DC_HANDLE_POOL_CODE);
            }
          else
            DC_problem.DC_nsubproblems=newint;
        }
    break;
    case 'p' :
      if(sscanf(arg,"-p%d",&newint) == 1)
        {
          if (newint<1)
            {
              fprintf(stderr,_DC_HANDLE_NUMCHROMO);
              exit(_DC_HANDLE_NUMCHROMO_CODE);
            }
          else
            {
              numchromo=newint;
            }
        }
      break;
    case 'N':               /* Non-interactive Run */
      interactive = FALSE;
      if(strlen(arg) < 4)
        {
          fprintf(stderr,_DC_HANDLE_INTERACTIVE);
          exit(_DC_HANDLE_INTERACTIVE_CODE);
        }
      switch(*(arg+2))
        {
        case 'o':
          finaldump = TRUE;
        case 'O':
          n_for_stop = 1;
          break;
        case 'a':
          finaldump = TRUE;
        case 'A':
          n_for_stop = -1; /* because numpop may not yet be set! */
          break;
        default:
          fprintf(stderr,_DC_HANDLE_INTERACTIVE_TYPE);
          exit(_DC_HANDLE_INTERACTIVE_CODE);
        }
      if(sscanf(arg+3,"%lf",&newdouble) == 1)
        {
          user_threshold = newdouble;
        }
      else
        {
          fprintf(stderr,_DC_HANDLE_NONINTERACTIVE_THRESHOLD);
          exit(_DC_HANDLE_INTERACTIVE_CODE);
```

```
      }
    break;
  case 'l' :        /* NO TENGO NI IDEA DE PARA QUE SIRVE */
    if(sscanf(arg,"-l%d",&newint) == 1)
      {
        if (newint<1)
          {
            fprintf(stderr,
          "DC_GA: Less than one generation per stage isn't sensible!\n");
            exit(1);
          }
        else
          {
            gens_stage=newint;
          }
      }
    break;
  case 'i' :        /* Reporting Interval */
    if(sscanf(arg,"-i%d",&newint) == 1)
      {
        if (newint<1)
          {
            fprintf(stderr,_DC_HANDLE_REPORT,newint);
            exit(_DC_HANDLE_REPORT_CODE);
          }
        else
          {
            repinterval=newint;
          }
      }
    break;
  case 'S':        /* Set the Random Seed */
    if(sscanf(arg,"-S%d",&newint) == 1)
      {
        if(newint<0)
          {
            fprintf(stderr,_DC_HANDLE_SEED);
            newint = 1234567890;
          }
        randseed = newint;
      }
    break;
  case 'a':       /* Adaptive Mutation */
    isadaptive=!isadaptive;
    if (isadaptive)
      mrate=adaptive;
    else
      mrate=nonadaptive;
    break;
  case 'F':      /* Datafile Selection */
    datafilename=strdup(arg+2);
    if(strlen(datafilename) == 0)
      {
        fprintf(stderr,_DC_HANDLE_DATAFILE);
        exit(_DC_HANDLE_DATAFILE_CODE);
      }
    break;

  case 'A':
    if(!strcmp(arg,"-Arandom"))
```

```
          assignment_type=_DC_RandomAssignment;
        else
          if(!strcmp(arg,"-Aseq"))
            assignment_type=_DC_SequentialAssignment;
          else
            {
              fprintf(stderr,_DC_HANDLE_ASSINGMENT,arg+2);
              exit(_DC_HANDLE_ASSINGMENT_CODE);
            }
        break;
      case 'm' :      /* Sets mutation rate */
        if(sscanf(arg,"-m%lf",&newdouble) == 1)
          {
            if (newdouble<0 || newdouble>1)
              {
                fprintf(stderr,_DC_HANDLE_MUTERATE);
                exit(_DC_HANDLE_MUTERATE_CODE);
              }
            else
              {
                mute_rate=newdouble;
              }
          }
        break;
      case 'c' :     /* Sets crossover rate */
        if(sscanf(arg,"-c%lf",&newdouble) == 1)
          {
            if (newdouble<0 || newdouble>1)
              {
                fprintf(stderr,_DC_HANDLE_CROSSRATE);
                exit(_DC_HANDLE_CROSSRATE_CODE);
              }
            else
              {
                cross_rate=newdouble;
              }
          }
        break;
      case 'b' :   /* Sets bias for rank based selection: select.c */
    if(sscanf(arg,"-b%lf",&newdouble) == 1)
      {
        if (newdouble<1.0 || newdouble>2.0)
          {
            fprintf(stderr,_DC_HANDLE_BIAS);
            exit(_DC_HANDLE_BIAS_CODE);
          }
        else
          {
            bias=newdouble;
          }
      }
        break;

/*
      case 'j' :       ******* Crowding Factor *******
        sscanf(arg,"-j%d",&crowding_factor);
        if(crowding_factor<1)
          {
            fprintf(stderr,
                    "DC_GA: Crowding factor must be larger than 1.\n");
```

```
            exit(1);
          }
        break;
*/

      case 'I':      /* Sets Initilasation Tournament Size */
        sscanf(arg,"-I%d",&newint);
        if(newint<1)
          {
            fprintf(stderr,_DC_HANDLE_INITTOUR);
            exit(_DC_HANDLE_INITTOUR_CODE);
          }
        else
          scp_init_mutation_tournament=newint;
        break;


      case 'M':
        if(!strcmp(arg,"-Msingle"))
          {
            mutate_name="single_bit";
            mutatemulti=FALSE;
          }
        else
          if(!strcmp(arg,"-Mmulti"))
            {
              mutate_name="multi_bit";
              mutatemulti=TRUE;
            }
          else
            {
              fprintf(stderr,_DC_HANDLE_MUTATION,arg+2);
              exit(_DC_HANDLE_MUTATION_CODE);
            }
        break;


      case 's' :     /* Set Scored Selection Function */
        if (!strcmp(arg,"-srank"))
          {
            selection_type=_DC_ScoredSelection;
            selection = rank_select;
            select_name="rank";
          }
        else if (!strcmp(arg,"-sfitprop"))
          {
            selection_type=_DC_ScoredSelection;
            selection = fitprop_select;
            select_name="fitprop";
          }
        else if (!strncmp(arg,"-stm",4))
          {
            selection_type=_DC_ScoredSelection;
            selection = tm_select;
            select_name=strdup(arg+2);
            tournament_size = atoi(arg+4);
            if (tournament_size < 1)
              {
                fprintf(stderr,_DC_HANDLE_TMARG);
                tournament_size = 10;
```

```
          }
      }
    else if (!strncmp(arg,"-stn",4))
      {
        selection_type=_DC_ScoredSelection;
        selection = tn_select;
        select_name=strdup(arg+2);
        tournament_size = atoi(arg+4);
        if (tournament_size < 1)
          {
            fprintf(stderr,_DC_HANDLE_TNARG);
            tournament_size = 10;
          }
      }
    else if (!strcmp(arg,"-srandom"))
      {
        selection_type=_DC_RandomSelection;
        selection = tn_select;
        select_name="tn";
        tournament_size=3;
      }
    else
      {
        fprintf(stderr,_DC_HANDLE_SELECTION,arg+2);
        exit(_DC_HANDLE_SELECTION_CODE);
      }
    break;
  case 'r' :       /* Select Reproduction type */
    if (!strcmp(arg,"-rone"))
      {
        reproduction = one_reproduction;
        repro_name="steady";
      }
    else if (!strcmp(arg,"-ronecross"))
      {
        reproduction = one_cross_reproduction;
        repro_name="simple_steady";
      }
    else
      {
        fprintf(stderr,_DC_HANDLE_REPROD,arg+2);
        exit(_DC_HANDLE_REPROD_CODE);
      }
    break;
  case 'C':           /* Select Crossover operator */
    if (!strcmp(arg,"-Cone"))
      {
        cross_name="one-point";
        crossover = one_pt_cross;
      }
    else if (!strcmp(arg,"-Ctwo"))
      {
        cross_name="two-point";
        crossover = two_pt_cross;
      }
    else if (!strcmp(arg,"-Cuniform"))
      {
        cross_name="uniform";
        crossover = uniform_cross;
      }
```

```
    else if (!strcmp(arg,"-Cnone"))
      {
        cross_name="none";
        crossover = no_cross;
      }
    break;
  case 'e' :
    if(!strncmp(arg,"-escp",5))
      {
        if(!(nerror = read_scp_file())==0)
          {
            if(nerror==_SCP_FILE_OPEN_CODE)
              {
                fprintf(stderr,_SCP_FILE_OPEN);
                exit(_SCP_FILE_OPEN_CODE);
              }
            if(nerror==_SCP_FILE_MALLOC_CODE)
              {
                fprintf(stderr,_SCP_FILE_MALLOC);
                exit(_SCP_FILE_MALLOC_CODE);
              }
          }
        eval_name="dc_scp";
        scp=1;
        sprintf(maxfitness,"%s","unknown");
        scp_mutation_tournament = atoi(arg+5);
        isbdsp=0;
      }
    else
      if(!strncmp(arg,"-ebds",5))
        {
          if(!(nerror = read_scp_file())==0)
            {
              if(nerror==_SCP_FILE_OPEN_CODE)
                {
                  fprintf(stderr,_SCP_FILE_OPEN);
                  exit(_SCP_FILE_OPEN_CODE);
                }
              if(nerror==_SCP_FILE_MALLOC_CODE)
                {
                  fprintf(stderr,_SCP_FILE_MALLOC);
                  exit(_SCP_FILE_MALLOC_CODE);
                }
            }
          eval_name="dc_bds";
          scp=1;
          sprintf(maxfitness,"%s","unknown");
          scp_mutation_tournament = atoi(arg+5);
          isbdsp=1;
        }
      else
        {
          fprintf(stderr,_DC_HANDLE_EVAL_MUTATE);
          exit(_DC_HANDLE_EVAL_MUTATE_CODE);
        }
    break;
  default :
    if (*(arg+1)!='h')
      fprintf(stderr,_DC_HANDLE_INVALIDARG,arg);
    dcscpga_help();
```

```
          exit(0);
        }
    }
  else
    {                         /* ... or an output filename. */
      output=fopen(arg,"a");
      if (output==NULL)
        {
          fprintf(stderr,_DC_HANDLE_OUTFILE,arg);
          exit(_DC_HANDLE_OUTFILE_CODE);
        }
      strcpy(outfilename,arg);
    }
}


/**********************************************************************/
/* Dumps chromosomes to a file. Filename is outfilename.chr, eg foo.chr */
/**********************************************************************/
void dump_chromos(void)
{
  int npool,nchromo,ngene;

  if(!chromoutput)
    {
      sprintf(chromofilename,"%s.chr",outfilename);
      chromofile = fopen(chromofilename,"a");
      chromoutput = (chromofile != (FILE *)NULL);
    }

  if(chromoutput)
    {
      for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
        {
          for(nchromo=0;nchromo<numchromo;nchromo++)
            {
              fprintf(chromofile,"%g ",
                      (1/DC_solution[npool].population[nchromo].fitness)-1.0);

              for(ngene=0;ngene<DC_subproblem[npool].size;ngene++)
                fprintf(chromofile," %d ",
                        DC_solution[npool].population[nchromo].gdp
                        ->genotype[ngene]-'0');
              fprintf(chromofile,"\n");
            }
          fprintf(chromofile,"\n\n");
        }
    }
}

main(int argc,char *argv[])
{
  int command,chromo,pop,gen,mignext,action,gen_total,fileoutput,i,j;
  int npool,nchromo,row,ncover;
  int *cover;
  double average;
  char c;
  int mapped, saved, converged, satisfied;
  CHROMOSOME *pops,migchromo;
  FILE *cfp;
```

```
randseed = (int) time(0);

DC_problem.DC_nsubproblems=5;
mignext=0;
n_for_stop = 0;
satisfied = FALSE;
geno_size_string = (char *)malloc(20);
sprintf(geno_size_string,"%d",geno_size);
maxfitness=geno_size_string;
eval_name="dc_scp";
cross_name="two-point";
select_name="tn3";
repro_name="simple_steady";
mutate_name="single_bit";
datafilename="scpdata";

if(argc==1)
    handle("-h");
else
  {
    for (command=1; command<argc; command+=1)
      handle(argv[command]);                    /* Handle any arguments. */
  }
srandom(randseed);

 if(scp)
  {
    if(strlen(outfilename)==0)
      sprintf(coverfilename,"%s.cov","cover");
    else
      sprintf(coverfilename,"%s.cov",outfilename);
    coverfile=fopen(coverfilename,"a");
    if(coverfile==(FILE *)NULL)
      {
        fprintf(stderr,
                "\nDC_GA_Evaluation: Error Openning Best Cover File.");
        exit(-1);
      }
  }

if(interactive || n_for_stop < 0) /* If interactive, stop test is that ALL */
                                  /* have converged; user may say sooner.. */
  n_for_stop = DC_problem.DC_nsubproblems;


/* Check if the user provided a filename for output. */
fileoutput = (output == (FILE *)NULL)? FALSE : TRUE;

if(interactive)
  {
    stdscr = initscr();
    last_line = tgetnum("li");
    if(last_line < 0) last_line = 24;

    mvprintw(VER_ROW,VER_COL,TITLE);
    mvprintw(VER_ROW,VER_COL+strlen(TITLE),": %s",datafilename);
    mvprintw(POP_ROW,POP_COL,      "        Subproblem Pools: %-6d",
             DC_problem.DC_nsubproblems);
    mvprintw(CPP_ROW,CPP_COL,      "  Chromosomes per Pool: %-4d",numchromo);
    mvprintw(EVLIM_ROW,EVLIM_COL,  "Generations per stage: %-6d",gens_stage);
```

```
    mvprintw(REPINT_ROW,REPINT_COL,
             "   Reporting interval: %-6d",repinterval);
    mvprintw(CTYPE_ROW,CTYPE_COL,  "   Crossover type: %s",cross_name);
    if(!strcmp(repro_name,"simple_steady"))
      mvprintw(CROSS_ROW,CROSS_COL,"   Crossover rate: n/a");
    else
      mvprintw(CROSS_ROW,CROSS_COL,"   Crossover rate: %3.2f",cross_rate);
    if(!strcmp(select_name,"rank"))
      mvprintw(BIAS_ROW,BIAS_COL,  "   Selection bias: %3.2f",bias);
    else
      mvprintw(BIAS_ROW,BIAS_COL,  "   Selection bias: n/a");
    if(isadaptive)
      mvprintw(MUT_ROW,MUT_COL,    "    Mutation: %s adaptive %.4f",
               mutate_name,mute_rate);
    else
      mvprintw(MUT_ROW,MUT_COL,    "    Mutation: %s %.6f",mutate_name,
               mute_rate);

    if(fileoutput)
      mvprintw(FNAME_ROW,FNAME_COL,"    Log file name: %s",outfilename);

    mvprintw(EVAL_ROW,EVAL_COL,    "Eval function: %s",eval_name);
    mvprintw(SELECT_ROW,SELECT_COL,"    Selection: %s",select_name);
    mvprintw(REPRO_ROW,REPRO_COL,  " Reproduction: %s",repro_name);
    mvprintw(POPTABLE_ROW,POPTABLE_COL,
             "Pool.......Average..........Best.(Min = %s)",maxfitness);
    mvprintw(GEN_ROW,GEN_COL,"Generation: ");
    clrtoeol();
    mvprintw(EVCOUNT_ROW,EVCOUNT_COL,"Evaluations so far: ");
    clrtoeol();
    refresh();
    cbreak();
    noecho();
  }

total_abs_fitness = (double *)malloc(DC_problem.DC_nsubproblems*
                                     sizeof(double));
initialise(numchromo);
gen_total = 1;
evals=0;
action = CONTINUE;

if(fileoutput) {
  fprintf(output,   "\n\n=====================================\n");
  fprintf(output,TITLE);
  fprintf(output,  "\n            data_file = %s\n",datafilename);
  fprintf(output,   "                 eval = %s\n",eval_name);
  fprintf(output,   "          max-fitness = %s\n",maxfitness);
  fprintf(output,   "         reproduction = %s\n",repro_name);
  fprintf(output,   "               select = %s\n",select_name);
  if(!strcmp(select_name,"rank"))
    fprintf(output, "       selection-bias = %.6f\n",bias);
  else
    fprintf(output, "       selection-bias = n/a\n");
  fprintf(output,   "                Pools = %d\n",
          DC_problem.DC_nsubproblems);
  fprintf(output,   "Chromosome # per pool = %d\n",numchromo);
  fprintf(output,   "       crossover-type = %s\n",cross_name);
  if(!strcmp(repro_name,"simple_steady"))
    fprintf(output, "       crossover-rate = n/a\n");
```

```
  else
    fprintf(output, "       crossover-rate = %.6f\n",cross_rate);
  if(isadaptive)
    fprintf(output, "        mutation-rate = adaptive\n");
  else
    fprintf(output, "        mutation-rate = %.6f\n", mute_rate);
  fprintf(output,   "   reporting-interval = %d\n", repinterval);
  fprintf(output,   "   random-number-seed = %d\n", randseed);
  fprintf(output,   "\n");
}

converged = report(0,fileoutput,user_threshold);
saved=FALSE;
converged = 0; /* DWC -- to allow continuation after phenotype convergence */
i = 0; /* used to gather up any digits typed in */

while(action != QUIT)
  {
    if(interactive)
      do
        {
          setprompt(fileoutput,saved,satisfied);
          mvprintw(0,1,prompt);
          clrtoeol();
          refresh();
          switch(c=getchar())
            {
            case 'a': case 'A':
              printw("starting again..");
              refresh();
              move(GEN_ROW,GEN_COL+12);
              clrtoeol();
              move(EVCOUNT_ROW,EVCOUNT_COL+20);
              clrtoeol();
              refresh();
              free_pops();
              gen_total = 1;
              initialise(numchromo);
              evals = 0;
              if(fileoutput)
                {
                  fprintf(output,
                          "\n\n------- re-starting -------\n\n");
                }
              converged=report(0,fileoutput,user_threshold);
              converged = 0; /* DWC */
              satisfied = (converged >= n_for_stop)? TRUE:FALSE;
              saved = FALSE;
              action = ASK_AGAIN;
              i = 0; /* reset number input */
              break;
            case 'c': case 'C':
              if(converged == DC_problem.DC_nsubproblems)
                action = ASK_AGAIN;
              else
                {
                  printw("continuing.."); refresh();
                  action = CONTINUE;
                  saved = FALSE;
                  if(i>0)
```

```
                  {
                    gens_stage = i;
                    mvprintw(EVLIM_ROW,EVLIM_COL,
                             "Generations per stage: %-6d",
                             gens_stage);
                    i=0;
                  }
              }
            break;
          case 'q': case 'Q':
            action = QUIT;
            break;
          case 's': case 'S':
            if(fileoutput && !saved)
              {
                printw("saving chromosomes.."); refresh();
                dump_chromos();
                saved = TRUE;
              }
            else if(!saved)
              {
                strcpy(outfilename,"unnamed");
                printw("saving chromosomes in unnamed.chr..");
                refresh();
                dump_chromos();
                saved = TRUE;
              }
            action = ASK_AGAIN;;
            i = 0; /* reset number input */
            break;
          case '0':case '1':case '2':case '3':case '4':
          case '5':case '6':case '7':case '8':case '9':
            i = 10*i + c - '0';
            action = ASK_AGAIN;
            break;
          default:
            action = ASK_AGAIN;
            i = 0; /* reset number input */
            break;
          }
      }
    while(action == ASK_AGAIN);
  if(action == CONTINUE)
    {
      for(gen=1; gen <= gens_stage && !satisfied; gen++, gen_total++)
        {
          reproduction();
          if (gen_total%repinterval==0)
            {                      /* Report */
              converged = report(gen_total,fileoutput,user_threshold);
              converged = 0; /* DWC */
            }
          if(converged >= n_for_stop)
            satisfied = TRUE;
        }
      if((gen_total-1)%repinterval != 0)
        { /* Last report a while ago? ... */
          converged =
            report(gen_total-1,fileoutput,user_threshold); /* ..report. */
          converged = 0; /* DWC */
```

```
            if(converged >= n_for_stop)
              satisfied = TRUE;
          }
        if(!interactive)
          {
            if(fileoutput && finaldump)
              dump_chromos();
            action = QUIT;
          }
      }
    }

  /* All done, tidy up */
  if(interactive)
    {
      echo();
      nocbreak();
      move(last_line-1,0);
      refresh();
      endwin();
    }

  if(fileoutput)
    fclose(output);
  if(chromoutput)
    fclose(chromofile);


  if(scp)
    fclose(coverfile);

  free_pops();
  exit(0);
}
```

```
/*

*****************************************************************************
**                                                                         **
**          MSc Information Technology: Knowledge Based Systems            **
**                Divide and Conquer Genetic Algorithms                    **
**                    for the Set-Covering Problem                         **
**                                                                         **
**   Module:     Credit Assignment                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    Mon Jun 19 21:00:00 1995                                  **
**   Last Modified: Sun Sep 10 18:22:19 1995                               **
**                                                                         **
**   Routines:   dc_random_selection: selects a chromojoint randomly       **
**               dc_scored_selection: selects a chromojoint according to the **
**                                    fitness of its members               **
**               dc_getgenotype: returns the genotype in a chromosome      **
**               dc_getcoverage: returns the cover corresponding to some   **
**                               gene                                      **
**               dc_indextorow:  returns the row in some given position    **
**               dc_eval_scp: evaluates a complete cover                   **
**               scorecover:  assigns a score to a cover compared to the   **
**                            best sofar                                   **
**               intcompare:  compares two integers for sorting            **
**               dc_random_peer_selection: selects the rest of the chunks  **
**                                    randomly.                            **
**               dc_scored peer_selection: selects the rest of the chunks  **
**                                    according to their fitness.          **
**               dc_evaluate_solution: evaluates the current solution      **
**                                     population                          **
**                                                                         **
**   External Data Structures:                                             **
**      int selection_type: how the other chunks are selected to evaluate  **
**                          a new chromosome.                              **
**      int geno_size: size of the genotype in a chromosome                **
**      int evals: number of evaluations already carried out               **
**      CHROMOSOME (*selection): pointer to the current selection function **
**      char *coverfilename: string containing the cover filename          **
**      char *datafilename:  string containing the data filename           **
**      FILE *coverfile: pointer to the cover file                         **
**                                                                         **
*****************************************************************************

*/

#include "dcscpga.h"
#include "errormsg.h"
#include "prototypes.h"
#include <stdio.h>
#include <stdlib.h>

/* #define _DEBUG_PRINTOUTS_ */
extern FILE *coverfile;
extern CHROMOSOME (*selection)(CHROMOSOME *,int *,int, int);

/****************************************************************************
**                                                                         **
**   Module:     Credit Assignment                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
```

```
**   Purpose:    Randomly selects one individual per pool for evaluation     **
**   Arguments:  -                                                           **
**   Global Data Structures:                                                 **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution         **
**                                 populations.                              **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                    data.                                  **
**   Remarks: -                                                              **
**                                                                           **
*****************************************************************************/
CHROMOJOINT dc_random_selection()
{
  int index;
  int value;
  CHROMOJOINT selected;
  selected.chunks=(CHROMOSOME *)
    malloc(DC_problem.DC_nsubproblems*sizeof(CHROMOSOME));
  selected.whatchunks=(int *)
    malloc(DC_problem.DC_nsubproblems*sizeof(int));
  if((selected.chunks==(CHROMOSOME *)NULL)||
     (selected.whatchunks==(int *)NULL))
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_DC_SELECT_MALLOC);
#endif
      exit(_DC_SELECT_MALLOC_CODE);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      value=(int)(drandom()*(double)(DC_solution[index].nmembers));
      selected.whatchunks[index]=value;
      selected.chunks[index]=DC_solution[index].population[value];
    }
  return selected;
}

/*****************************************************************************
**                                                                           **
**   Module:     Credit Assignment                                           **
**   Author:     Luis F. Gonzalez Hernandez                                  **
**   Created:    June 1995                                                   **
**   Purpose:    Selects one individual per pool for evaluation according    **
**               to its fitness.                                             **
**   Arguments:  -                                                           **
**   Global Data Structures:                                                 **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution         **
**                                 populations.                              **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                    data.                                  **
**   Remarks: -                                                              **
**                                                                           **
*****************************************************************************/
CHROMOJOINT dc_scored_selection()
{
  int index;
    CHROMOJOINT selected;
  selected.chunks=(CHROMOSOME *)
    malloc(DC_problem.DC_nsubproblems*sizeof(CHROMOSOME));
  selected.whatchunks=(int *)
    malloc(DC_problem.DC_nsubproblems*sizeof(int));
```

```
  if((selected.chunks==(CHROMOSOME *)NULL)||
     (selected.whatchunks==(int *)NULL))
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_DC_SELECT_MALLOC);
#endif
      exit(_DC_SELECT_MALLOC_CODE);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      geno_size = DC_subproblem[index].size;
      selected.chunks[index]=
        selection(DC_solution[index].population,&(selected.whatchunks[index]),
                  index,DC_solution[index].nmembers);
    }
  return selected;
}

/*****************************************************************************
**                                                                         **
**   Module:     Credit Assignment                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Given the current Chromojoint being evaluated and the     **
**               number of the row, returns the genotype value.            **
**   Arguments:  -                                                         **
**   Global Data Structures:                                               **
**   DC_ROW  DC_subproblem - structure problem data                        **
**   Remarks: -                                                            **
**                                                                         **
*****************************************************************************/
int dc_getgenotype(CHROMOJOINT chromo,int index)
{
  int counter=0;
  while(index>=DC_subproblem[counter].size)
    index-=DC_subproblem[counter++].size;
  return (chromo.chunks[counter].gdp)->genotype[index];
}

/*****************************************************************************
**                                                                         **
**   Module:     Credit Assignment                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Given the gene and the number of the row, translates the  **
**               gene, returning the corresponding coverer.                **
**   Arguments:  -                                                         **
**   Global Data Structures:                                               **
**   DC_ROW  DC_subproblem - structure problem data                        **
**   Remarks: -                                                            **
**                                                                         **
*****************************************************************************/
int dc_getcoverage(int index,int gene)
{
  int counter=0;
  while(index>=DC_subproblem[counter].size)
    index-=DC_subproblem[counter++].size;
  return DC_subproblem[counter].dc_row_coverage[index].coverers[gene];
}
```

```
/*************************************************************************
**                                                                     **
**   Module:     Credit Assignment                                     **
**   Author:     Luis F. Gonzalez Hernandez                            **
**   Created:    June 1995                                             **
**   Purpose:    Translates the order in the solution to the row.      **
**   Arguments:  -                                                     **
**   Global Data Structures:                                           **
**   DC_ROW  DC_subproblem - structure problem data                    **
**   Remarks: Note this is necessary because of the random pool assignment   **
**                                                                     **
*************************************************************************/
int dc_indextorow(int index)
{
  int counter=0;
  while(index>=DC_subproblem[counter].size)
    index-=DC_subproblem[counter++].size;
  return DC_subproblem[counter].rows[index];
}

/*************************************************************************
**                                                                     **
**   Module:     Credit Assignment                                     **
**   Author:     Luis F. Gonzalez Hernandez                            **
**   Created:    June 1995                                             **
**   Purpose:    Evaluates the cover provided by one Chromojoint and returns **
**               the fitness.                                          **
**   Arguments:  Pointer to a CHROMOJOINT to be evaluated              **
**   Global Data Structures:  char isbdsp Flag to indicate whether or not    **
**                                      the problem is of bus-drivers.     **
**   DC_ROW  DC_subproblem - structure problem data                    **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution       **
**                                 populations.                        **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                   data.                             **
**   Remarks:    Memory is allocated just once at the beginning to avoid     **
**               delays.                                               **
**                                                                     **
*************************************************************************/
double dc_eval_scp(CHROMOJOINT *chromo)
{
  static double fitness_best_sofar=0.0;
  static int ncover_best_sofar=0,*best_cover_sofar;
  double fitness,score;
  int index,subindex,row;
  int ncols,current,column,not_suspect,cost=0;
  int joint,intersection;
  static char *covered, *active;
  static int *cover_count, *cols;
  static allocated=0;
  if(!allocated)
    {
      best_cover_sofar=(int *)malloc(DC_problem.scp_rows*sizeof(int));
      covered=(char *)malloc(DC_problem.scp_rows*sizeof(char));
      active=(char *)malloc(DC_problem.scp_rows*sizeof(char));
      cover_count=(int *)malloc(DC_problem.scp_rows*sizeof(int));
      cols=(int *)malloc(DC_problem.scp_rows*sizeof(int));
      allocated=1;
    }
  for(index=0;index<DC_problem.scp_rows;index++)
```

```
    covered[index]=active[index]=cover_count[index]=0;
  ncols=0;
  for(index=0;index<DC_problem.scp_rows;index++)
    {
      row=dc_indextorow(index);
      /*
         Activate the Gene if not covered by any other set already.
         */
      if(covered[row]==0)
        {
          active[row]=1;
          current=dc_getgenotype(*chromo,index)-'0';
          column=dc_getcoverage(index,current);
          cost+=DC_problem.scp_costs[column];
          for(subindex=0;subindex<DC_col_coverage[column].ncovers;subindex++)
            {
              covered[DC_col_coverage[column].covers[subindex]]=1;
              cover_count[DC_col_coverage[column].covers[subindex]]++;
            }
          cols[ncols++]=column;
        }
    }
  /*
     The cover is defined by the columns indicated in the active genes.
     Before the cost is assesed, we try to get rid of redundant columns
     (That is, those columns that are 'subsumed' by combinations of
     others).
     */
  for(index=0;index<(DC_problem.scp_rows-1);index++)
    {
      row=dc_indextorow(index);
      if(active[row])
        {
          current=dc_getgenotype(*chromo,index)-'0';
          column=dc_getcoverage(index,current);
          not_suspect=0;
          for(subindex=0;subindex<DC_col_coverage[column].ncovers;subindex++)
            if(cover_count[DC_col_coverage[column].covers[subindex]]==1)
              {
                not_suspect=1;
                break;
              }
          if(!not_suspect) /* Column is redundant */
            {
              active[row]=0;
              for(subindex=0;subindex<DC_col_coverage[column].ncovers;
                  subindex++)
                cover_count[DC_col_coverage[column].covers[subindex]]--;
              cost-=DC_problem.scp_costs[column];
              for(subindex=0;subindex<ncols;subindex++)
                if(cols[subindex]==column)
                  {
                    cols[subindex]=cols[ncols-1];
                    ncols--;
                    break;
                  }
            }
        }
    }
  qsort((char *)cols,ncols,sizeof(int),intcompare);
```

```
  chromo->ncover=ncols;

  /*
     The important factor for Bus Driver scheduling Problems is not
     how good the cost of the cover is but how many sets it is made
     of. So the cost is just a by-product that allows to make a between
     two covers with the same number of sets.
     */
  if(isbdsp)
    fitness=(10.0/(double)(ncols))+(1.0/(double)(1+cost));
  else
    fitness=1/(double)(1+cost);

  /*
     Calculate the best cover sofar.
     */

  if(fitness_best_sofar<=fitness)
    {
      joint=0;
      intersection=0;
      score=scorecover(cols,ncols,best_cover_sofar,ncover_best_sofar,
                       &joint,&intersection);
      if((int)(score*10000)!=0)
        {
          fprintf(coverfile,
                  "\nScore: %g\tCost: %d\tSets: %d\n",score,cost,ncols);
          ncover_best_sofar=ncols;
          fitness_best_sofar=fitness;
          for(index=0;index<ncols;index++)
            {
              best_cover_sofar[index]=cols[index];
              fprintf(coverfile," %d",best_cover_sofar[index]);
            }
          fprintf(coverfile,"\n");
          for(index=ncols;index<DC_problem.DC_nsubproblems;index++)
            best_cover_sofar[index]=0;
        }
    }
  return(fitness);
}

/******************************************************************************
**                                                                          **
**   Module:     Credit Assignment                                          **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Purpose:    Calculates the score of a new cover comparing it with the  **
**               previous best cover found.                                 **
**   Arguments:  int *newcover: Pointer to the new cover                    **
**               int nnew: Number of sets in the new cover                  **
**               int *oldcover: Pointer to the old cover                    **
**               int nold: Number of sets in the old cover                  **
**               int *joint: Pointer to the number of elements in the union  **
**                           of the covers.                                 **
**               int *intersection: Pointer to the number of elements in the **
**                                  intersection.                           **
**   Global Data Structures: -                                              **
**   Remarks: the score is calculated as the ratio between the difference of **
**            union and intersection to the union alone.                    **
```

```
**                                                                          **
****************************************************************************/
double scorecover(int *newcover,int nnew,int *oldcover,int nold,
               int *joint,int *intersection)
{
  int indexnew,indexold;
  *joint=nnew+nold;
  *intersection=0;
  for(indexnew=0;indexnew<nnew;indexnew++)
    for(indexold=0;indexold<nold;indexold++)
      if(oldcover[indexold]>newcover[indexnew])
        break;
      else
        if(oldcover[indexold]==newcover[indexnew])
          {
            (*joint)--;
            (*intersection)++;
          }
  if(*joint<=0)
    return -1;
  else
    return (double)((*joint)-(*intersection))/(double)(*joint);
}

/****************************************************************************
**                                                                          **
**   Module:     Credit Assignment                                          **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Purpose:    Compare two integers for ordering through qsort.           **
**   Arguments:  integer1: Pointer to the first integer in the comparasion  **
**               integer2: Pointer to the second integer in the comparasion **
**   Global Data Structures: -                                              **
**   Remarks: -                                                             **
**                                                                          **
****************************************************************************/
static int intcompare(const void *integer1, const void *integer2)
{
  return(*((int *)integer1)-*((int *)integer2));
}

/****************************************************************************
**                                                                          **
**   Module:     Credit Assignment                                          **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Purpose:    Randomly selects one individual per pool to evaluate one   **
**               chromosome.                                                **
**   Arguments:  int pool: Pool to which the chromosome being evaluated     **
**                         belongs.                                         **
**               int nchromo: Number of the chromosome being evalutated     **
**                            within the population                         **
**               CHROMOSOME newchunk: chromosome being evaluated            **
**   Global Data Structures:                                                **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution        **
**                                 populations.                             **
**   struct problem_data DC_problem - structure that holds problem          **
**                                    definition data.                      **
**   Remarks: -                                                             **
**                                                                          **
```

```
****************************************************************************/
CHROMOJOINT dc_random_peer_selection(int pool,int nchromo,CHROMOSOME newchunk)
{
  int index;
  int value;
  CHROMOJOINT selected;
  selected.chunks=(CHROMOSOME *)
    malloc(DC_problem.DC_nsubproblems*sizeof(CHROMOSOME));
  selected.whatchunks=(int *)
    malloc(DC_problem.DC_nsubproblems*sizeof(int));
  if((selected.chunks==(CHROMOSOME *)NULL)||
     (selected.whatchunks==(int *)NULL))
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_DC_SELECT_MALLOC);
#endif
      exit(_DC_SELECT_MALLOC_CODE);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      if(index==pool)
        {
          selected.whatchunks[index]=nchromo;
          selected.chunks[index]=newchunk;
        }
      else
        {
          value=(int)(drandom()*(double)(DC_solution[index].nmembers));
          selected.whatchunks[index]=value;
          selected.chunks[index]=DC_solution[index].population[value];
        }
    }
  return selected;
}

/****************************************************************************
**                                                                         **
**   Module:     Credit Assingment                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Selects one individual per pool to evaluate one chromosome **
**               according to its fitness                                  **
**   Arguments:  int pool: Pool to which the chromosome being evaluated    **
**                         belongs.                                        **
**               int nchromo: Number of the chromosome being evalutated    **
**                            within the population                        **
**               CHROMOSOME newchunk: chromosome being evaluated           **
**   Global Data Structures:                                               **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution       **
**                                 populations.                            **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                    data.                                **
**   Remarks: -                                                            **
**                                                                         **
****************************************************************************/
CHROMOJOINT dc_scored_peer_selection(int pool,int nchromo,CHROMOSOME newchunk)
{
  int index;
    CHROMOJOINT selected;
```

```
  selected.chunks=(CHROMOSOME *)
    malloc(DC_problem.DC_nsubproblems*sizeof(CHROMOSOME));
  selected.whatchunks=(int *)
    malloc(DC_problem.DC_nsubproblems*sizeof(int));
  if((selected.chunks==(CHROMOSOME *)NULL)||
     (selected.whatchunks==(int *)NULL))
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_DC_SELECT_MALLOC);
#endif
      exit(_DC_SELECT_MALLOC_CODE);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      if(index==pool)
        {
          selected.whatchunks[index]=nchromo;
          selected.chunks[index]=newchunk;
        }
      else
        {
          geno_size = DC_subproblem[index].size;
          selected.chunks[index]=
            selection(DC_solution[index].population,
                      &(selected.whatchunks[index]),
                      index,DC_solution[index].nmembers);
        }
    }
  return selected;
}

/*****************************************************************************
**                                                                         **
**   Module:     Credit Assignment                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Evaluates the solution in DC_solution by sequentially     **
**               picking up one chromosome and making sure it has been     **
**               evaluated in a CHROMOJOINT at least                       **
**               _MIN_EVALUATIONS_THRESHOLD times.                         **
**   Arguments:  int numchromo: Number of Chromosomes in each pool.        **
**   Global Data Structures:                                               **
**   DC_ROW  DC_subproblem - structure problem data                        **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution       **
**                                 populations.                            **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                    data.                                **
**   Remarks:    Memory is allocated just once at the beginning to avoid   **
**               delays.                                                   **
**                                                                         **
*****************************************************************************/
void dc_evaluate_solution(int numchromo)
{
  static int allocated=0;
  int index,position,poolstart,subindex;
  int npool,evaluate;
  double fitness_score;
  int totaver=0;
  CHROMOSOME  current;
  CHROMOJOINT selected;
```

```
  struct eval_data
    {
      double accum_fitness;
      int nevals;
    };
  static struct eval_data *evaldata;

  evals++;
   if(!allocated)
     {
      evaldata=(struct eval_data *)
        malloc(numchromo*
               DC_problem.DC_nsubproblems*sizeof(struct eval_data));
      if(evaldata==(struct eval_data *)NULL)
        {
#ifdef _DEBUG_PRINTOUTS_
          fprintf(stderr,_DC_SELECT_MALLOC);
#endif
          exit(_DC_SELECT_MALLOC_CODE);
        }
       allocated=1;
     }
  /*
     First initilise the values of the temporal structre evaldata
     */
  for(index=0;index<numchromo*DC_problem.DC_nsubproblems;index++)
    {
      evaldata[index].nevals=0;
      evaldata[index].accum_fitness=0.0;
    }

  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,"Entering into evaluation of pool: %d",npool);
#endif
      for(evaluate=0;evaluate<numchromo;evaluate++)
        {
#ifdef _DEBUG_PRINTOUTS_
          fprintf(stderr,"Selecting chromo #%d: %d",evaluate);
#endif
          while(evaldata[npool*numchromo+evaluate].nevals
                <_MIN_EVALUATIONS_THRESHOLD)
            {
              current=DC_solution[npool].population[evaluate];
              if(selection_type==_DC_RandomSelection)
                selected=dc_random_peer_selection(npool,evaluate,current);
              else
                selected=dc_scored_peer_selection(npool,evaluate,current);

              position=0;
              poolstart=0;

              fitness_score=dc_eval_scp(&selected);

              for(index=0;index<DC_problem.DC_nsubproblems;index++)
                {
                  position=index*DC_solution[index].nmembers+
                    selected.whatchunks[index];
                  evaldata[position].nevals++;
```

```
                evaldata[position].accum_fitness+=fitness_score;
              }
            free(selected.whatchunks);
            free(selected.chunks);
          }
      }
  }

  /*
     Now that the evaluations have been carried_out
     we will assign the scores to the individuals.
     */
  position=0;
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      for(subindex=0;subindex<DC_solution[index].nmembers;
          subindex++,position++)
        {
          if(evaldata[position].nevals<1)
            {
#ifdef _DEBUG_PRINTOUTS_
              printf("\nPool: %d\tPosition: %d",index,position);
#endif
              DC_solution[index].population[subindex].fitness=0.0;
            }
          else
            {
              DC_solution[index].population[subindex].fitness=
                evaldata[position].accum_fitness/
                  (double)evaldata[position].nevals;
#ifdef _DEBUG_PRINTOUTS_
              if(DC_solution[index].population[subindex].fitness>1)
                {
                  printf("\nAnomalous Result:");
                  printf("\nPool: %d\tChromo:%d\tEvals:%d\tFitness:%f",index,
                         subindex,evaldata[position].nevals,
                         DC_solution[index].population[subindex].fitness);
                }
#endif
            }
        }
    }
}
```

```
/*

*****************************************************************************
**                                                                         **
**           MSc Information Technology: Knowledge Based Systems           **
**                 Divide and Conquer Genetic Algorithms                   **
**                     for the Set-Covering Problem                        **
**                                                                         **
**   Module:     Initialisation                                            **
**   Routines:                                                             **
**  int read_scp_file(int ,int)             Reads the data file            **
**  int dc_create_subproblems(int, int *)    Creates the datastructures that **
**                                          will hold the D&C information  **
**  void dc_fixedsize_pools(int, int, int *)  Sizes of the pools are fixed  **
**  void dc_variablesize_pools(int, int, int *) Sizes of the pools are      **
**                                             variable                    **
**  int dc_assign_random_values(int *)       Does the random assignment of  **
**                                          the elements to the pools      **
**  int dc_sort_compare(const void *,const void *) Comparison function used  **
**                                               with qsort in the random  **
**                                               assingment                **
**                                                                         **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Last Modified: Wed Sep 13 13:00:01 1995                               **
**                                                                         **
**   Routines in other modules:                                            **
**   Global Data Structures:                                               **
**                                                                         **
*****************************************************************************

*/
/*

Testing code for function: read_scp_file()

#define _DEBUG_PRINTOUTS_
int read_scp_file();

int assignment_type;
char *datafilename;
main(int argc, char *argv[])
{
  datafilename=SCPDATAFILE;
  switch(argc)
    {
    case 1:
      assignment_type==_DC_RandomAssignment;
      DC_problem.DC_nsubproblems=2;
      srandom(1234567890);
      break;
    case 4:
      assignment_type=(int)(*argv[1]-'0');
      DC_problem.DC_nsubproblems=(int)(*argv[2]-'0');
      srandom((atoi(argv[3])));
      break;
    default:
      printf("\nTo use this program you should either supply three or");
      printf("\nno arguments. The arguments are:");
      printf("\nType of Assignment (0-Random/1-Sequential) [default:0]");
```

```
      printf("\nNumber of Subproblems [default:2]");
      printf("\nRandom Number Seed [default:1234567890]\n");
      return -1;
    }
  printf("\nTesting datafile reading:");

  switch(read_scp_file(1))
    {
    case _SCP_FILE_OPEN_CODE:
      exit(_SCP_FILE_OPEN_CODE);
      break;
    case _SCP_FILE_MALLOC_CODE:
      exit(_SCP_FILE_MALLOC_CODE);
      break;
    case 0:
    default:
      printf("\nTesting finished.");
    }
}
*/

/* #define _DEBUG_PRINTOUTS_ */

#ifndef MaxSizeOsc
#define MaxSizeOsc (10)
#endif
#ifndef SCPDATAFILE
#define SCPDATAFILE "scpdata"
#endif

#include <stdio.h>
#include "dcscpga.h"
#include "errormsg.h"
#include "prototypes.h"
#include <string.h>

/*
   Auxiliar Functions to read data in, created by Peter Ross and
   defined in readdata.c
*/
double getdouble(FILE *, double, int);
int getint(FILE *, int *, int);
int getlong(FILE *, long *, int);

extern char *datafilename;
extern int assignment_type;

/*****************************************************************************
**                                                                         **
**   Module:  Initialisation                                               **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Reads the data file in.                                   **
**   Arguments:  int subproblems: Number of subproblem pools               **
**               int assignment_type: Random or Sequential Assignment      **
**                                                                         **
**   Global Data Structures:                                               **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                  data.                                  **
**   DC_ROW  DC_subproblem - structure problem data                        **
```

```
**   SCP_COL *DC_col_coverage -  Pointer to each of the structures          **
**                               that carry the information about column    **
**                               covering                                   **
**   Remarks: Intended to be used with the standard qsort function.         **
**                                                                          **
****************************************************************************/
int read_scp_file(int subproblems,int assignment_type)
{
  FILE *file;
  int *order;
  int index,subindex,counter=0;
#ifdef _DEBUG_PRINTOUTS_
  int subsubindex;
#endif
  int intTemp;

  if((file=fopen(datafilename,"r"))==(FILE *)NULL)
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_SCP_FILE_OPEN);
#endif
      return _SCP_FILE_OPEN_CODE;
    }
  else
    {
      getint(file,&(DC_problem.scp_rows),FALSE); /* SCP Rows Number */
      getint(file,&(DC_problem.scp_cols),FALSE); /* SCP Cols Number */

      /*
         Assign memory to the data structures that we are going to use
         to hold the problem data for each subproblem.
       */
      DC_subproblem=(DC_ROW *)
        malloc(DC_problem.DC_nsubproblems*sizeof(DC_ROW));
      if(DC_subproblem==NULL)
        {
#ifdef _DEBUG_PRINTOUTS_
          fprintf(stderr,_SCP_FILE_MALLOC);
#endif
          return _SCP_FILE_MALLOC_CODE;
        }
      /* Get the order in which the elements should be assigned to
         the pools */
      order=(int *)malloc(DC_problem.scp_rows*sizeof(int));
      if(order == NULL)
        {
#ifdef _DEBUG_PRINTOUTS_
          fprintf(stderr,_SCP_FILE_MALLOC);
#endif
          return _SCP_FILE_MALLOC_CODE;
        }
      dc_create_subproblems(assignment_type,order);
      /*
         Reading the costs into Memory:
         1. Memory allocation for the costs
         2. Read the values from the file
      */
      DC_problem.scp_costs=(int *)malloc(DC_problem.scp_cols*sizeof(int));
      if(DC_problem.scp_costs == NULL)
        {
```

```
#ifdef _DEBUG_PRINTOUTS_
          fprintf(stderr,_SCP_FILE_MALLOC);
#endif
          free(DC_subproblem);
          free(order);
          return _SCP_FILE_MALLOC_CODE;
        }
      DC_col_coverage=(SCP_COL *)
        malloc(DC_problem.scp_cols*sizeof(SCP_COL));
      if(DC_col_coverage==NULL)
        {
#ifdef _DEBUG_PRINTOUTS_
          fprintf(stderr,_SCP_FILE_MALLOC);
#endif
          free(order);
          free(DC_problem.scp_costs);
          return _SCP_FILE_MALLOC_CODE;
        }
      /* Reading the costs and Initialising DC_col_coverage */
      for(index=0;index<DC_problem.scp_cols;index++)
        {
          getint(file,&intTemp,FALSE);
          DC_problem.scp_costs[index]=intTemp;
          DC_col_coverage[index].ncovers=0;
          DC_col_coverage[index].covers=NULL;
        }

      /* Reading the overall structure */
        for(index=0;index<DC_problem.scp_rows;index++)
          {
            while(order[index]>=DC_subproblem[counter].size)
              order[index]-=DC_subproblem[counter++].size;
            DC_subproblem[counter].rows[order[index]]=index;
            getint(file,&intTemp,FALSE);
            DC_subproblem[counter].dc_row_coverage[order[index]].ncoverers
              =intTemp;

            printf("\nReading Row[%d].",index);
            fflush(stdout);

            DC_subproblem[counter].dc_row_coverage[order[index]].coverers=
              (int *)malloc
                (DC_subproblem[counter].dc_row_coverage[order[index]].ncoverers
                 *(sizeof(int)));
            if(DC_subproblem[counter].dc_row_coverage[order[index]].coverers
               ==NULL)
            {
#ifdef _DEBUG_PRINTOUTS_
              fprintf(stderr,_SCP_FILE_MALLOC);
#endif
              return _SCP_FILE_MALLOC_CODE;
            }
              for(subindex=0;subindex<
                DC_subproblem[counter].dc_row_coverage[order[index]].ncoverers;
                  subindex++)
                {
                  getint(file,&intTemp,FALSE);
        DC_subproblem[counter].dc_row_coverage[order[index]].coverers[subindex]
          =(intTemp-1);
                  DC_col_coverage[(intTemp-1)].covers=(int *)
```

```
                    realloc(DC_col_coverage[(intTemp-1)].covers,
                            (DC_col_coverage[(intTemp-1)].ncovers+1)
                            *sizeof(int));
                  if(DC_col_coverage[(intTemp-1)].covers==NULL)
                    {
#ifdef _DEBUG_PRINTOUTS_
                      fprintf(stderr,_SCP_FILE_MALLOC);
#endif
                      return _SCP_FILE_MALLOC_CODE;
                    }
                  DC_col_coverage[(intTemp-1)].
                    covers[DC_col_coverage[(intTemp-1)].ncovers++]=index;
                }
            counter=0;
          }
    }

#ifdef _DEBUG_PRINTOUTS_
  printf("\nData read from file:");
  printf("\nSCP Rows: %d",DC_problem.scp_rows);
  printf("\nSCP Cols: %d",DC_problem.scp_cols);
  for(index=0;index<DC_problem.scp_cols;index++)
    {
      printf("\nCosts of Column %d:\t%d",index,
             DC_problem.scp_costs[index]);
      for(subindex=0;subindex<DC_col_coverage[index].ncovers;subindex++)
        printf("\nColumn %d\tcovers Row: %d",index,
               DC_col_coverage[index].covers[subindex]);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      printf("\nPool: %d",index);
      for(subindex=0;subindex<DC_subproblem[index].size;subindex++)
        {
          printf("\nRow: %d\tNumber of Coverers: %d",
                 DC_subproblem[index].rows[subindex],
                 DC_subproblem[index].dc_row_coverage[subindex].ncoverers);

          for(subsubindex=0;
              subsubindex<DC_subproblem[index].dc_row_coverage[subindex]
              .ncoverers;subsubindex++)
            printf("\nCoverer #%d\tSet: %d",subsubindex,
         DC_subproblem[index].dc_row_coverage[subindex].coverers[subsubindex]);
        }
    }
#endif
  fclose(file);
  return 0;
}

int dc_create_subproblems(int assginment_type,int *order)
{
  int index;
  int *poolsizes;
  struct translate
    {
      int size;
      int *order;
    };
  /*
```

```
     First we are going to determine the allocation of the elements
     within the pools:

     1. Get the sizes of the pools
     2. Do allocation

  */
  poolsizes=(int *)malloc(DC_problem.DC_nsubproblems*sizeof(int));
  if(poolsizes==NULL)
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_SCP_FILE_MALLOC);
#endif
      return _SCP_FILE_MALLOC_CODE;
    }
  /* Get the sizes of the pools */

 if(assignment_type==_DC_RandomAssignment)
   dc_fixedsize_pools(DC_problem.scp_rows,DC_problem.DC_nsubproblems,
                      poolsizes);
 else
   dc_variablesize_pools(DC_problem.scp_rows,DC_problem.DC_nsubproblems,
                      poolsizes);
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      DC_subproblem[index].size=poolsizes[index];
      DC_subproblem[index].dc_row_coverage=
        (SCP_ROW *)malloc(DC_subproblem[index].size*sizeof(SCP_ROW));
      if(DC_subproblem[index].dc_row_coverage==NULL)
      {
#ifdef _DEBUG_PRINTOUTS_
        fprintf(stderr,_SCP_FILE_MALLOC);
#endif
        return _SCP_FILE_MALLOC_CODE;
      }
      DC_subproblem[index].rows=
        (int *)malloc(DC_subproblem[index].size*sizeof(int));
      if(DC_subproblem[index].rows==NULL)
      {
#ifdef _DEBUG_PRINTOUTS_
        fprintf(stderr,_SCP_FILE_MALLOC);
#endif
        return _SCP_FILE_MALLOC_CODE;
      }
    }
  free(poolsizes);
  /*
     Do the allocation:
     If we have random assingment, get the random order or else get the
     sequential order.
   */
  if(assignment_type==_DC_RandomAssignment)
    {
      if((dc_assign_random_values(order))!=0)
        return _SCP_FILE_MALLOC_CODE;
    }
  else
      for(index=0;index<DC_problem.scp_rows;index++)
        order[index]=index;
#ifdef _DEBUG_PRINTOUTS_
```

```
  printf("\n Printing the assingment order.");
  for(index=0;index<DC_problem.scp_rows;index++)
    printf("\nRow: %d\tOrder: %d",index,order[index]);
#endif

}

/*****************************************************************************
**                                                                         **
**   Module:  Initialisation                                               **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Compares two chromosomes returning one if the second is   **
**               fitter than the first, zero if equally fit and minus one  **
**               otherwise.                                                **
**   Arguments:  int totalsize Size of the Set Covering Problem (Rows)     **
**               int npools    Number of pools                             **
**               int *size     Pointer to a matrix with the sizes of each  **
**                             pool.                                       **
**   Global Data Structures: -                                             **
**   Remarks: -                                                            **
**                                                                         **
*****************************************************************************/
void dc_fixedsize_pools(int totalsize,int npools,int *size)
{
  int index;
  int counter;
  counter=totalsize;
  for(index=1;index<npools;index++)
    {
      size[index]=(int)totalsize/
        (int)npools;
      counter-=size[index];
    }
  *size=counter;
}

/*****************************************************************************
**                                                                         **
**   Module:  Initialisation                                               **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Compares two chromosomes returning one if the second is   **
**               fitter than the first, zero if equally fit and minus one  **
**               otherwise.                                                **
**   Arguments:  int totalsize Size of the Set Covering Problem (Rows)     **
**               int npools    Number of pools                             **
**               int *size     Pointer to a matrix with the sizes of each  **
**                             pool.                                       **
**   Global Data Structures: -                                             **
**   Remarks: Intended to be used to favour the sequential assignment      **
**                                                                         **
*****************************************************************************/
void dc_variablesize_pools(int totalsize, int npools, int *size)
{
  dc_fixedsize_pools(totalsize,npools,size); /* Right Now this doesn't work */
}

/*****************************************************************************
**                                                                         **
```

```
**   Module:  Initialisation                                                 **
**   Author:     Luis F. Gonzalez Hernandez                                  **
**   Created:    June 1995                                                   **
**   Purpose:    Compares two chromosomes returning one if the second is     **
**               fitter than the first, zero if equally fit and minus one    **
**               otherwise.                                                  **
**   Arguments:  void const *arg: Pointer to the first record                **
**               void const *arg: Pointer to the second record               **
**   Global Data Structures: -                                               **
**   Remarks: Intended to be used with the standard qsort function.          **
**                                                                           **
*****************************************************************************/
int dc_sort_compare(const void *arg1,const void *arg2)
{
  struct record
    {
      int random_value;
      int original;
    };
  struct record *first, *second;
  first=(struct record *)arg1;   /* To avoid a warning message issued by */
  second=(struct record *)arg2;  /* the compiler. */
  return (first->random_value - second->random_value);
}

/*****************************************************************************
**                                                                           **
**   Module:  Initialisation                                                 **
**   Author:     Luis F. Gonzalez Hernandez                                  **
**   Created:    June 1995                                                   **
**   Purpose:    This function creates the data-structures that hold the     **
**               problem information using the given order                   **
**   Arguments:  int *order Pointer to a matrix holding the orden in which   **
**                          the elements are assigned to the pools           **
**   Global Data Structures:                                                 **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                    data.                                  **
**   Remarks: -                                                              **
**                                                                           **
*****************************************************************************/
int dc_assign_random_values(int *order)
{
  struct record
    {
      int random_value;
      int original;
    };
  int index;
  int *random_values;
  struct record *group;
  group=(struct record *)malloc(DC_problem.scp_rows*sizeof(struct record));
  if(group==NULL)
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_SCP_FILE_MALLOC);
#endif
      return _SCP_FILE_MALLOC_CODE;
    }
  for(index=0;index<DC_problem.scp_rows;index++)
    {
```

```
      group[index].random_value=(int)((double)DC_problem.scp_rows*drand48());
      group[index].original=index;
    }
  qsort((struct record *)group,DC_problem.scp_rows,
        sizeof(struct record),dc_sort_compare);  /* Quicksort from the
                                                    standard Library    */
  for(index=0;index<DC_problem.scp_rows;index++)
    order[index]=group[index].original;
  free(group);
  return 0;
}
```

```
/*
******************************************************************************
**                                                                          **
**           MSc Information Technology: Knowledge Based Systems            **
**                  Divide and Conquer Genetic Algorithms                   **
**                      for the Set-Covering Problem                        **
**                                                                          **
**   Module:  Initialisation                                                **
**   Author:  Luis F. Gonzalez                                              **
**   Created:    1993                                                       **
**   Last Modified: Sun Sep 10 18:23:39 1995                                **
**   Routines:   compchromo: used to compare two chromosomes when ordered   **
**                           using the standard qsort function.             **
**               init_genotype: generates the genotype of a chromosome      **
**               init_pops_sorted: creates and sorts according the fitness  **
**                                 all pools in a population.               **
**                                                                          **
**   External Data Structures:                                              **
**      int scp_init_mutation_tournament: Initialisation tournament size    **
**      int chromo_id: Counter of the last unique chromosome id.            **
**      double *total_abs_fitness: Pointer to a matrix holding the total    **
**                                 absolute fitness per pool.               **
**                                                                          **
**   Remark: Based heavily on init.c original from Peter Ross and Geoff     **
**           Ballinger (c) 1993                                             **
**                                                                          **
******************************************************************************
*/

#include <stdio.h>
#include <math.h>
#include <malloc.h>

#include "dcscpga.h"
#include "prototypes.h"
#include "errormsg.h"

#define MINIMUMCOST_INIT (1000000.0)

extern int scp_init_mutation_tournament;
extern int chromo_id;
extern double *total_abs_fitness;
extern long random();


/*****************************************************************************
**                                                                          **
**   Module:  Initialisation                                                **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Purpose:    Compares two chromosomes returning one if the second is    **
**               fitter than the first, zero if equally fit and minus one   **
**               otherwise.                                                 **
**   Arguments:  void const *chromo1: Pointer to the first chromsome        **
**               void const *chromo2: Pointer to the second chromosome      **
**   Global Data Structures: -                                              **
**   Remarks: Intended to be used with the standard qsort function.         **
**                                                                          **
```

```
****************************************************************************/
int compchromo(void const *chromo1,void const *chromo2)
{
  if((((CHROMOSOME *)chromo2)->fitness) >
    (((CHROMOSOME *)chromo1)->fitness))
     return(1);
  else
    if((((CHROMOSOME *)chromo2)->fitness) ==
       (((CHROMOSOME *)chromo1)->fitness))
      return(0);
    else
      return(-1);
}


/****************************************************************************
**                                                                        **
**   Module:  Initialisation                                              **
**   Author:     Luis F. Gonzalez Hernandez                               **
**   Created:    June 1995                                                **
**   Purpose:    Generates the chromosome's gentotype string. Each allele  **
**               is created using a tournament. The best in the tournament **
**               (from randomly picked coverers) is selected. If several   **
**               the same score, one is randomly taken.                    **
**   Arguments:  int pool: Pool number to which the Chromosome belongs     **
**               GENODATA *gdp: Pointer to the genotype string.            **
**   Global Data Structures:                                              **
**   struct problem_data DC_problem - structure that hols problem definition **
**                                  data.                                 **
**   DC_ROW  DC_subproblem - structure problem data                       **
**   int scp_initialisation_tournament                                    **
**   SCP_COL *DC_col_coverage -  Pointer to each of the structures         **
**                               that carry the information about column  **
**                               covering                                 **
**   Remarks: -                                                           **
**                                                                        **
****************************************************************************/
void init_genotype(int pool,GENODATA *gdp)
{
  int index,counter,choice,best,nummin,coverer;
  double minc;

  gdp->refcount = 1;
  gdp->id = chromo_id++;
  gdp->parent1 = gdp->parent2 = 0;
  gdp->genotype = (int *)malloc((DC_subproblem[pool].size+1)*sizeof(int));
  gdp->genotype[DC_subproblem[pool].size] = '\0';  /* for printing */

  for(index=0;index<DC_subproblem[pool].size;index++)
    {
      minc=MINIMUMCOST_INIT;
      best=0;

      /*
         The scp_init_mutation_tournament must be smaller than the number
         of rows to be covered.
         */
      for(counter=0;counter<scp_init_mutation_tournament;counter++)
        {
         choice = (random()%DC_subproblem[pool].dc_row_coverage[index]
```

```
                 .ncoverers);
          coverer = DC_subproblem[pool].dc_row_coverage[index]
            .coverers[choice];
          if(((double)DC_problem.scp_costs[coverer]/
              (double)DC_col_coverage[coverer].ncovers)<minc)
            {
              minc=((double)DC_problem.scp_costs[coverer]/
                    (double)DC_col_coverage[coverer].ncovers);
              nummin=1;
              best=choice;
            }
          else
            if(((double)DC_problem.scp_costs[coverer]/
                (double)DC_col_coverage[coverer].ncovers)==minc)
              {
                nummin++;
                if(drandom()<(1.0/nummin)) best=choice;
              }
        }
      gdp->genotype[index] = '0'+best;
    }
}

/*****************************************************************************
**                                                                         **
**   Module:  Initialisation                                               **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Creates and Initialises the DC_solution structure         **
**               that contains the solution population.                    **
**   Arguments:  int numchromo: Number of Chromosomes in the Pools         **
**   Global Data Structures:                                               **
**   struct problem_data DC_problem - structure that holds problem         **
**                                     definition data.                    **
**   DC_ROW  DC_subproblem - structure problem data                        **
**   Remarks: -                                                            **
**                                                                         **
*****************************************************************************/
void init_pops_sorted(int numchromo)
{
  int npool,chromo;
  CHROMOSOME *pops, *chromp;

  DC_solution=(SUBPROBLEM_POOL *)
    malloc(DC_problem.DC_nsubproblems*sizeof(SUBPROBLEM_POOL));
  if(DC_solution == (SUBPROBLEM_POOL *)NULL)
    {
      fprintf(stderr,_DC_INIT_MALLOC);
      exit(1);
    }

  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    {
      /*
         Total_abs_fitness is used in fitness proportional selection and
         needs to be calculated therefore for each pool independently.
         */
      total_abs_fitness[npool]=0.0;
      DC_solution[npool].nmembers=numchromo;
      DC_solution[npool].population=(CHROMOSOME *)
```

```
        malloc(DC_solution[npool].nmembers*sizeof(CHROMOSOME));
      if(DC_solution[npool].population==(CHROMOSOME *)NULL)
        {
          fprintf(stderr,_DC_INIT_MALLOC);
          exit(1);
        }

      /* Now we initialise the populations */
      for(chromo=0; chromo<numchromo;chromo++)
        {
          DC_solution[npool].population[chromo].gdp =
            (GENODATA *)malloc(sizeof(GENODATA)); /* pops.     */

          /* Initialise the Genotype */
          init_genotype(npool,DC_solution[npool].population[chromo].gdp);
        }
    }

  /*
     Once all Genotypes have been initialised, proceed onto allocating
     the initial fitness for the popultation.
     */
  dc_evaluate_solution(numchromo);

      /*
         Total_abs_fitness is used in fitness proportional selection and
         needs to be calculated therefore for each pool independently.
         */
  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    for(chromo=0; chromo<numchromo;chromo++)
      total_abs_fitness[npool] +=
          fabs(DC_solution[npool].population[chromo].fitness);

   for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
     qsort((CHROMOSOME *)(DC_solution[npool].population),
           DC_solution[npool].nmembers,sizeof(CHROMOSOME),compchromo);
}
```

```
/*

******************************************************************************
**                                                                          **
**           MSc Information Technology: Knowledge Based Systems            **
**                 Divide and Conquer Genetic Algorithms                    **
**                     for the Set-Covering Problem                         **
**   Module:     Reproduction                                               **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Last Modified: Wed Sep 13 12:33:42 1995                                **
**                                                                          **
**   Routines:   one_cross_reproduction: Steady State Genetic Algorithm     **
**                                       that forces the use of crossover   **
**                                       with probability 1.                **
**               one_reproduction: Steady State GA that allows the use of   **
**                                 a crossover rate                         **
**                                                                          **
**   Remarks:    Based heavily on the version by Peter Ross and Geoff       **
**               Ballinger (c) 1993, 1994                                   **
**                                                                          **
******************************************************************************

*/
#include "dcscpga.h"
#include "errormsg.h"
#include "prototypes.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/*
   Pointers to the functions used.
   */
extern CHROMOSOME (*selection)(CHROMOSOME *,int *,int, int);
extern CHROMOSOME (*crossover)(CHROMOSOME, CHROMOSOME, int);
extern char *mutate_name;
extern double (*mrate)(CHROMOSOME,CHROMOSOME,int);
/* extern double (*evaluate)(); */

#define _EVALUATE_POPULATION_INTERVAL (5)

/*
   Global variables accessed.
   */
extern double mute_rate;
extern double cross_rate;
extern double *total_abs_fitness;
extern int evals;
extern int selection_type;
extern int mutatemulti;

/*****************************************************************************
**                                                                          **
**   Module:     Reproduction                                               **
**   Author:     Luis F. Gonzalez Hernandez                                 **
**   Created:    June 1995                                                  **
**   Purpose:    Carries out a steady state divide and conquer GA.          **
**   Arguments:  -                                                          **
**   Global Data Structures:                                                **
```

```
**               selection_type: Records whether Random or Scored selection  **
**                               is being used.                              **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution         **
**                                 populations.                              **
**   Remarks:    Based heavily on the version by Peter Ross and Geoff        **
**               Ballinger (c) 1993, 1994                                    **
**                                                                           **
*****************************************************************************/
void one_cross_reproduction(void)
{
  CHROMOSOME parent1, parent2;
  CHROMOJOINT child,selected;
  CHROMOSOME offspring;
  static int pop_eval_interval=1;
  int which_chromo;
  int index,npool,neval,chromo;
  double *child_fitness,m_rate;

  child_fitness=(double *)malloc(DC_problem.DC_nsubproblems*sizeof(double));
  child.whatchunks=(int *)malloc(DC_problem.DC_nsubproblems*sizeof(int));
  child.chunks=(CHROMOSOME *)malloc(DC_problem.DC_nsubproblems*
                                  sizeof(CHROMOSOME));
  if((child.whatchunks==(int *)NULL)||
     (child.chunks==(CHROMOSOME *)NULL)||
     (child_fitness==(double *)NULL))
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_DC_REPROD_MALLOC);
#endif
      exit(_DC_REPROD_MALLOC_CODE);
    }

  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    {
      child_fitness[npool]=0.0;
      parent1=selection(DC_solution[npool].population,&which_chromo,
                        npool,DC_solution[npool].nmembers);
      parent2=selection(DC_solution[npool].population,&which_chromo,
                        npool,DC_solution[npool].nmembers);
      offspring=crossover(parent1,parent2,npool);

      if(mutatemulti)
        child.chunks[npool]=mutate_multi(offspring,
                                         mrate(parent1,parent2,npool),
                                         npool);
      else
        child.chunks[npool]=mutate_one_bit(offspring,
                                           mrate(parent1,parent2,npool),
                                           npool);

      /*
         Evaluate the fitness of each chunk.
         */
      evals++;
      for(neval=0;neval<_AVERAGE_EVALUATIONS;neval++)
        {
          if(selection_type==_DC_RandomSelection)
            selected=dc_random_peer_selection(npool,_Offspring_Chunk,
                                              child.chunks[npool]);
          else
```

```
          selected=dc_scored_peer_selection(npool,_Offspring_Chunk,
                                           child.chunks[npool]);
         child_fitness[npool]+=dc_eval_scp(&selected);
         free(selected.chunks);
         free(selected.whatchunks);
        }
      child_fitness[npool]/=(double)(_AVERAGE_EVALUATIONS);
      child.chunks[npool].fitness=child_fitness[npool];
    }

  /*
     Crowding.
     */

  /*
     Insert the chunks into the population.
     */
  insert(child,child_fitness);
  free(child_fitness);
  free(child.chunks);
  free(child.whatchunks);

  if(pop_eval_interval++%_EVALUATE_POPULATION_INTERVAL)
    {
      pop_eval_interval=0;
      dc_evaluate_solution(DC_solution[0].nmembers);
      for(index=0;index<DC_problem.DC_nsubproblems;index++)
        {
          total_abs_fitness[index]=0.0;
          for(chromo=0; chromo<DC_solution[index].nmembers;chromo++)
            total_abs_fitness[index] +=
              fabs(DC_solution[index].population[chromo].fitness);
        }

      for(index=0;index<DC_problem.DC_nsubproblems;index++)
        qsort((CHROMOSOME *)(DC_solution[index].population),
              DC_solution[index].nmembers,sizeof(CHROMOSOME),compchromo);
    }
}


/*****************************************************************************
**                                                                         **
**   Module:     Reproduction                                              **
**   Author:     Luis F. Gonzalez Hernandez                                **
**   Created:    June 1995                                                 **
**   Purpose:    Carries out a steady state divide and conquer GA.         **
**   Arguments:  -                                                         **
**   Global Data Structures:                                               **
**               selection_type: Records whether Random or Scored selection **
**                               is being used.                            **
**   SUBPROBLEM_POOL DC_solution - structure that holds the solution       **
**                                 populations.                            **
**   Remarks:    Based heavily on the version by Peter Ross and Geoff      **
**               Ballinger (c) 1993, 1994                                  **
**                                                                         **
*****************************************************************************/
void one_reproduction(void)
{
  CHROMOSOME parent1, parent2;
```

```
  CHROMOJOINT child,selected;
  CHROMOSOME offspring;
  static int pop_eval_interval=1;
  int which_chromo;
  int index,npool,neval,chromo;
  double *child_fitness,m_rate;

  child_fitness=(double *)malloc(DC_problem.DC_nsubproblems*sizeof(double));
  child.whatchunks=(int *)malloc(DC_problem.DC_nsubproblems*sizeof(int));
  child.chunks=(CHROMOSOME *)malloc(DC_problem.DC_nsubproblems*
                                    sizeof(CHROMOSOME));
  if((child.whatchunks==(int *)NULL)||
     (child.chunks==(CHROMOSOME *)NULL)||
     (child_fitness==(double *)NULL))
    {
#ifdef _DEBUG_PRINTOUTS_
      fprintf(stderr,_DC_REPROD_MALLOC);
#endif
      exit(_DC_REPROD_MALLOC_CODE);
    }
  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    {
      child_fitness[npool]=0.0;
      parent1=selection(DC_solution[npool].population,&which_chromo,
                        npool,DC_solution[npool].nmembers);
      if(drandom()>cross_rate)
        offspring=parent1;
      else
        {
          parent2=selection(DC_solution[npool].population,&which_chromo,
                            npool,DC_solution[npool].nmembers);
          offspring=crossover(parent1,parent2,npool);
        }

      if(mutatemulti)
        child.chunks[npool]=mutate_multi(offspring,mute_rate,npool);
      else
        child.chunks[npool]=mutate_one_bit(offspring,mute_rate,npool);

      /*
         Evaluate the fitness of each chunk.
         */
      evals++;
      for(neval=0;neval<_AVERAGE_EVALUATIONS;neval++)
        {
          if(selection_type==_DC_RandomSelection)
            selected=dc_random_peer_selection(npool,_Offspring_Chunk,
                                              child.chunks[npool]);
          else
            selected=dc_scored_peer_selection(npool,_Offspring_Chunk,
                                              child.chunks[npool]);
          child_fitness[npool]+=dc_eval_scp(&selected);
          free(selected.chunks);
          free(selected.whatchunks);
        }
      child_fitness[npool]/=(double)(_AVERAGE_EVALUATIONS);
      child.chunks[npool].fitness=child_fitness[npool];
    }

  /*
```

```
    Crowding.
    */

  /*
    Insert the chunks into the population.
    */
  insert(child,child_fitness);
  free(child_fitness);

  if(pop_eval_interval++%_EVALUATE_POPULATION_INTERVAL)
    {
      pop_eval_interval=0;
      dc_evaluate_solution(DC_solution[0].nmembers);
      for(index=0;index<DC_problem.DC_nsubproblems;index++)
        {
          total_abs_fitness[index]=0.0;
          for(chromo=0; chromo<DC_solution[0].nmembers;chromo++)
            total_abs_fitness[index] +=
              fabs(DC_solution[index].population[chromo].fitness);
        }

      for(index=0;index<DC_problem.DC_nsubproblems;index++)
        qsort((CHROMOSOME *)(DC_solution[index].population),
              DC_solution[index].nmembers,sizeof(CHROMOSOME),compchromo);
    }
}

/****************************************************************************
**                                                                         **
**   Module:     Reproduction                                              **
**   Author:     Luis F. Gonzalez Hernandez (see Remarks)                  **
**   Created:    June 1995                                                 **
**   Purpose:    This function inserts 'chromo' into the correct place     **
**               (sorted by fitness) within the population pool defined    **
**   Arguments:                                                            **
**   Global Data Structures:                                               **
**   Remarks:    Based heavily on the version by Peter Ross and Geoff      **
**               Ballinger (c) 1993, 1994                                  **
**                                                                         **
****************************************************************************/

void insert(CHROMOJOINT offspring,double *offspring_fitness)
{
  int npool;
  int current;

  for(npool=0;npool<DC_problem.DC_nsubproblems;npool++)
    {

      /*
         First we check to see if the child is worth inserting.
         */
      if (offspring_fitness[npool]>
          DC_solution[npool].population[DC_solution[npool].nmembers-1].fitness)
        {
          offspring.chunks[npool].gdp->refcount++;
          if(DC_solution[npool].population[DC_solution[npool].nmembers-1]
             .gdp->refcount == 1)
            {
              total_abs_fitness[npool] -=
```

```
        fabs(DC_solution[npool]
             .population[DC_solution[npool].nmembers-1].fitness);
      free(DC_solution[npool].population
           [DC_solution[npool].nmembers-1].gdp->genotype);
      free(DC_solution[npool].population[DC_solution[npool]
                                         .nmembers-1].gdp);
    }
  else
    DC_solution[npool].population[DC_solution[npool].nmembers-1]
      .gdp->refcount--;
  /*
     Now we are going through the whole population (that is already
     ordered) till we can insert the new offspring.

     int current will be the counter that holds the value of the
                 current member of the population being checked.
     */

  current=DC_solution[npool].nmembers-2;/* Start at 2nd last place. */
  do
    {
      /*
         Check if offspring is fitter and either push the current one
         down or insert the new one in the current position and finish.
         */
      if (offspring_fitness[npool]>
          DC_solution[npool].population[current].fitness)
        DC_solution[npool].population[current+1]=
          DC_solution[npool].population[current];
      else
        {
          total_abs_fitness[npool] = total_abs_fitness[npool]
            + fabs(offspring_fitness[npool])
              - fabs(DC_solution[npool].population[current+1].fitness);
          DC_solution[npool].population[current+1]=
            offspring.chunks[npool];
          current=0;
        }
      current-=1;
    }
  while (current>=0);    /* Stop the loop if the whole population has
                            been checked. */
  /*
     Finally we check if our offspring is the most fit.
     */
  if (offspring_fitness[npool]>
      DC_solution[npool].population[0].fitness)
    {
      total_abs_fitness[npool] = total_abs_fitness[npool]
        + fabs(offspring_fitness[npool])
          - fabs(DC_solution[npool].population[0].fitness);
      DC_solution[npool].population[0]=offspring.chunks[npool];
    }
}
else
  {
  /*
     The offspring produced cannot be inserted because it is terrible.
     */
  if(offspring.chunks[npool].gdp->refcount==0)
```

```
            {
              free(offspring.chunks[npool].gdp->genotype);
              free(offspring.chunks[npool].gdp);
            }
        }
    }
}
```

```
/* select.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 */

/* Modified by Luis F. Gonzalez
   Last Modified: Tue Sep  5 18:55:54 1995

Modifications:
All functions are included a new argument which a pointer to the number
of chromosome in the population that has been selected.

Argument base is substituted by argument pool which is the pool being
analysed.

*/

#include "dcscpga.h"
#include <stdio.h>

extern double bias;
extern double *total_abs_fitness;
extern int tournament_size;

extern double drandom();
extern double sqrt();
extern double fabs();

#define ANY ((int)(drandom() * (double)range))

/**********************************************************************/
/* This function performs rank based selection and is lifted from      */
/* Whitley's paper on GENITOR in icga 89. It selects from the specified */
/* subpopulation within 'pops'.                                         */
/**********************************************************************/

CHROMOSOME rank_select(CHROMOSOME *pops, int *number, int pool, int range)
{
  double value;

  /* Argument error checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to work with pool %d.\n",
              pool);
      exit(-1);
    }
  if(range>DC_solution[pool].nmembers)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to use invalid range: %d.\n",
              range);
      exit(-1);
    }

  if(bias<1.05)
     value=drandom()/bias;
  else
    value=(bias-sqrt(bias*bias-4.0*(bias-1.0)*drandom()))/(2.0*(bias-1.0));
```

```
  value=value*(double)range;

  *number=(int)value;
  if(*number>range)
    {
      fprintf(stderr,
              "\nDC_GA_selection: Attempt to select invalid chromo: %d.\n",
              *number);
      exit(-1);
    }

  return(pops[*number]);
}


/**********************************************************************/
/* This function implements fitness proportionate selection. It treats  */
/* the fitness of the population as a "roulette wheel", rolls a ball on */
/* the wheel, and picks the chromosome within whose sector the ball     */
/* stops.                                                               */
/**********************************************************************/

CHROMOSOME fitprop_select(CHROMOSOME *pops, int *number, int pool, int range)
{
  int loop;
  double where;

  /* Argument error checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to work with pool %d.\n",
              pool);
      exit(-1);
    }
  if(range>DC_solution[pool].nmembers)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to use invalid range: %d.\n",
              range);
      exit(-1);
    }

  where=drandom()*total_abs_fitness[pool];  /* Random float in 0..total */

  for(loop=0; where>fabs(pops[loop].fitness); loop++)
    where-=fabs(pops[loop].fitness);

  *number=loop;
  if(*number>range)
    {
      fprintf(stderr,
              "\nDC_GA_selection: Attempt to select invalid chromo: %d.\n",
              *number);
      exit(-1);
    }

  return(pops[loop]);
}

/**********************************************************************/
/* This fitness function performs a form of tournament selection as in  */
```

```
/* the classic marriage problem in dynamic programming. Select one at   */
/* uniform random, then continue for up to tournament_size tries making */
/* uniform random selections: choose the first that exceeds the fitness */
/* of that first-chosen. After tournament_size failures, choose first.  */
/**********************************************************************/


CHROMOSOME tm_select(CHROMOSOME *pops, int *number, int pool, int range)
{
  int i, choice;
  double f;
  int reject = ANY;

  /* Argument error checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to work with pool %d.\n",
              pool);
      exit(-1);
    }
  if(range>DC_solution[pool].nmembers)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to use invalid range: %d.\n",
              range);
      exit(-1);
    }

  f = pops[reject].fitness;
  for(i=0;i<tournament_size;i++)
    {
      choice = ANY;
      if(pops[choice].fitness > f)
        {
          *number=choice;
          return(pops[choice]);
        }
    }

  *number=reject;
  if(*number>range)
    {
      fprintf(stderr,
              "\nDC_GA_selection: Attempt to select invalid chromo: %d.\n",
              *number);
      exit(-1);
    }

  return(pops[reject]);
}

/**********************************************************************/
/* This fitness function performs straight tournament selection as in   */
/* Brindle, Alberta TR 81-2. Pick tournament_size at uniform random,    */
/* return fittest of those.                                             */
/**********************************************************************/

CHROMOSOME tn_select(CHROMOSOME *pops, int *number, int pool, int range)
{
  int i, choice;
  double f;
```

```
  int one = ANY;

  /* Argument error checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to work with pool %d.\n",
              pool);
      exit(-1);
    }
  if(range>DC_solution[pool].nmembers)
    {
      fprintf(stderr,"\nDC_GA_selection: Attempt to use invalid range: %d.\n",
              range);
      exit(-1);
    }

  f = pops[one].fitness;
  for(i=0;i<tournament_size-1;i++)
    {
      choice = ANY;
      if(pops[choice].fitness > f)
        {
          one = choice;
          f = pops[one].fitness;
        }
    }

  *number=one;
  if(*number>range)
    {
      fprintf(stderr,
              "\nDC_GA_selection: Attempt to select invalid chromo: %d.\n",
              *number);
      exit(-1);
    }

  return(pops[one]);
}
```

```
/* cross.c
 * Copyright (C) 1993 Peter Ross and Geoff Ballinger.
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Crossover choices. All take two CHROMOSOMEs as arguments plus the
 * pool number in which the CHROMOSOMES are included and return
 * the crossed offspring. There are no twins, one of the resulting
 * CHROMOSOMES is lost.
 *
 * Modified by Luis F. Gonzalez to accept the new data structures defined
 * for the divide and conquer algorithm.
 *
 * Last Modified: Tue Sep  5 18:56:26 1995
 *
 */

/*
   Modification log:
   1. ANSI Function definition convention
   2. the mutation functions are added a new argument which is the
      pool number of the CHROMOSOME.
   3. geno_size is assigned the value DC_subproblem[pool].size


   Keep an eye on:

*/

#include <stdlib.h>
#include <stdio.h>
#include "dcscpga.h"
#include "errormsg.h"
#include "prototypes.h"

extern int chromo_id;

/*
   Testing code for module Crossover.
   */
/*
int geno_size;
int chromo_id=2;
int twins=0;

main()
{
  int genotype00[]={(int)'0',(int)'2',(int)'2'};
  int genotype01[]={(int)'0',(int)'2'};
  int genotype10[]={(int)'2',(int)'1',(int)'0'};
  int genotype11[]={(int)'0',(int)'2'};
  GENODATA chromodata0={ genotype00,1,1,0,0 };
  GENODATA chromodata1={ genotype10,1,1,0,0 };
  CHROMOSOME initial0,initial1,result;
  int npool,index,subindex,subsubindex;
  int costs[]={10,20,30,40,50,60};
  int colcov0[]={1,2,5};
  int colcov1[]={1,4,5};
  int colcov2[]={4,5};
  int colcov3[]={0,1,4,5};
```

```
int colcov4[]={0,1,2};
int colcov5[]={0,3};
int rowcov0[]={0,2};
int rowcov1[]={0,1,3};
int rowcov2[]={3,4};
int rowcov3[]={5};
int rowcov4[]={0,1,2,4};
SCP_COL col1={ 10,3,colcov0};
SCP_COL col2={ 20,3,colcov1};
SCP_COL col3={ 30,2,colcov2};
SCP_COL col4={ 40,4,colcov3};
SCP_COL col5={ 50,3,colcov4};
SCP_COL col6={ 60,2,colcov5};
SCP_ROW row0={2,rowcov0};
SCP_ROW row1={3,rowcov1};
SCP_ROW row2={2,rowcov2};
SCP_ROW row3={1,rowcov3};
SCP_ROW row4={4,rowcov4};

int pool0_row[]={0,1,2};
int pool1_row[]={3,4};

srandom((int)1234567890);
DC_problem.scp_rows=5;
DC_problem.scp_cols=6;
DC_problem.scp_costs=costs;

DC_col_coverage=(SCP_COL *)malloc(DC_problem.scp_cols*sizeof(SCP_COL));
DC_col_coverage[0]=col1;
DC_col_coverage[1]=col2;
DC_col_coverage[2]=col3;
DC_col_coverage[3]=col4;
DC_col_coverage[4]=col5;
DC_col_coverage[5]=col6;

DC_subproblem=(DC_ROW *)malloc(2*sizeof(DC_ROW));
DC_subproblem[0].size=3;
DC_subproblem[1].size=2;

DC_subproblem[0].rows=
  (int *)malloc(DC_subproblem[npool].size*sizeof(int));
DC_subproblem[0].dc_row_coverage=
  (SCP_ROW *)malloc(DC_subproblem[npool].size*sizeof(SCP_ROW *));
DC_subproblem[1].rows=
  (int *)malloc(DC_subproblem[npool].size*sizeof(int));
DC_subproblem[1].dc_row_coverage=
  (SCP_ROW *)malloc(DC_subproblem[npool].size*sizeof(SCP_ROW *));

DC_subproblem[0].rows=pool0_row;
DC_subproblem[1].rows=pool1_row;
DC_subproblem[0].dc_row_coverage[0]=row0;
DC_subproblem[0].dc_row_coverage[1]=row1;
DC_subproblem[0].dc_row_coverage[2]=row2;
DC_subproblem[1].dc_row_coverage[0]=row3;
DC_subproblem[1].dc_row_coverage[1]=row4;

printf("\nProblem Data:");
printf("\nSCP Rows: %d",DC_problem.scp_rows);
printf("\nSCP Cols: %d",DC_problem.scp_cols);
for(index=0;index<DC_problem.scp_cols;index++)
```

```
    {
      printf("\nCosts of Column %d:\t%d",index,
             DC_problem.scp_costs[index]);
      for(subindex=0;subindex<DC_col_coverage[index].ncovers;subindex++)
        printf("\nColumn %d\tcovers Row: %d",index,
               DC_col_coverage[index].covers[subindex]);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      printf("\nPool: %d",index);
      for(subindex=0;subindex<DC_subproblem[index].size;subindex++)
        {
          printf("\nRow: %d\tNumber of Coverers: %d",
                 DC_subproblem[index].rows[subindex],
                 DC_subproblem[index].dc_row_coverage[subindex].ncoverers);

          for(subsubindex=0;
              subsubindex<DC_subproblem[index].dc_row_coverage[subindex]
              .ncoverers;subsubindex++)
            printf("\nCoverer #%d\tSet: %d",subsubindex,
         DC_subproblem[index].dc_row_coverage[subindex].coverers[subsubindex]);
        }
    }

  initial0.gdp=&chromodata0;
  initial0.fitness=(double)0.5;
  initial0.nactive=initial0.ncover=0;
  initial1.gdp=&chromodata1;
  initial1.fitness=(double)0.5;
  initial1.nactive=initial1.ncover=0;

  for(index=0;index<10;index++)
    {
      printf("\nRun #%d",index);
      result=one_pt_cross(initial0,initial1,0);
      printf("\nOne point crossover GENOTYPE: ");
      for(subindex=0;subindex<DC_subproblem[0].size;subindex++)
        printf("%d ",result.gdp->genotype[subindex]-'0');

      result=two_pt_cross(initial0,initial1,0);
      printf("\nTwo point crossover GENOTYPE: ");
      for(subindex=0;subindex<DC_subproblem[0].size;subindex++)
        printf("%d ",result.gdp->genotype[subindex]-'0');

      result=uniform_cross(initial0,initial1,0);
      printf("\nUniform crossover GENOTYPE: ");
      for(subindex=0;subindex<DC_subproblem[0].size;subindex++)
        printf("%d ",result.gdp->genotype[subindex]-'0');

   }
  free(DC_subproblem[0].rows);
  free(DC_subproblem[1].rows);
  free(DC_col_coverage);
  free(DC_subproblem);
}
*/

/**********************************************************************/
/* This performs one point crossover on 'parent1' and 'parent2'.        */
/* Child is new (even though there may be identical genotypes elsewhere */
```

```
/* in the system) so has refcount 0; set to 1 only upon insertion      */
/**********************************************************************/

CHROMOSOME one_pt_cross(CHROMOSOME parent1,CHROMOSOME parent2,int pool)
{
  int chosen,i;
  CHROMOSOME child;

  /* Argument Error Checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,
              "\nDC_GA_Crossover: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }

  /* Generate crossover point randomly. */
  chosen=(int)(drandom() * DC_subproblem[pool].size);

  /* Create the child. */
  child.fitness=0.0;
  child.gdp = (GENODATA *)malloc(sizeof(GENODATA));
  child.gdp->refcount = 0;
  child.gdp->genotype = (int *)malloc((DC_subproblem[pool].size+1)*
                                      sizeof(int));
  child.gdp->genotype[DC_subproblem[pool].size]='\0';
  child.gdp->id = chromo_id++;
  child.gdp->parent1 = parent1.gdp->id;
  child.gdp->parent2 = parent2.gdp->id;

  /* Crossover */
  for(i=0; i<chosen; i++)
    child.gdp->genotype[i] = parent2.gdp->genotype[i];

  for(i=chosen; i<DC_subproblem[pool].size; i++)
    child.gdp->genotype[i] = parent1.gdp->genotype[i];

  return(child);
}

/**********************************************************************/
/* This performs two point crossover on 'parent1' and 'parent2'.       */
/* Child is new (even though there may be identical genotypes elsewhere */
/* in the system) so has refcount 0; set to 1 only upon insertion      */
/**********************************************************************/

CHROMOSOME two_pt_cross(CHROMOSOME parent1,CHROMOSOME parent2,int pool)
{
  int first,last,i,swap;
  CHROMOSOME child;

  /* Argument Error Checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,
              "\nDC_GA_Crossover: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }
```

```
  /* Generate the two points randomly. */
  first=(int)(drandom() * DC_subproblem[pool].size);
  last= (int)(drandom() * DC_subproblem[pool].size);

  if (first>last)
    {                                /* Make sure that they are in the */
      swap=first;                    /* right order.                   */
      first=last;
      last=swap;
    }

  /* Create the child. */
  child.fitness=0.0;
  child.gdp = (GENODATA *)malloc(sizeof(GENODATA));
  child.gdp->refcount = 0;
  child.gdp->genotype = (int *)malloc((DC_subproblem[pool].size+1)
                                    *sizeof(int));
  child.gdp->genotype[DC_subproblem[pool].size]='\0';
  child.gdp->id = chromo_id++;
  child.gdp->parent1 = parent1.gdp->id;
  child.gdp->parent2 = parent2.gdp->id;

  /* Do the crossover */
  for(i=0; i<first; i++)
      child.gdp->genotype[i] = parent2.gdp->genotype[i];

  for(i=first; i<last; i++)
      child.gdp->genotype[i] = parent1.gdp->genotype[i];

  for(i=last; i<DC_subproblem[pool].size; i++)
    child.gdp->genotype[i] = parent2.gdp->genotype[i];

  return(child);
}

/**********************************************************************/
/* This performs uniform crossover on 'parent1' and 'parent2'.        */
/* Uses rand_bit() to cut down on expensive random number generation. */
/* Child is new (even though there may be identical genotypes elsewhere */
/* in the system) so has refcount 0; set to 1 only upon insertion      */
/**********************************************************************/

int rand_bit_src;
int rand_bit_bits = 0;
int rand_bit()
{
  if(rand_bit_bits == 0)
    {
      rand_bit_bits = 16;

      /* Get 16+1 more random bits */
      rand_bit_src = (int)(drandom() * 131072.0);
    }
  rand_bit_src = rand_bit_src >> 1;   /* Lose 1 bit */
  rand_bit_bits--;                    /* Number left after this call */
  return(rand_bit_src & 1);
}


CHROMOSOME uniform_cross(CHROMOSOME parent1,CHROMOSOME parent2,int pool)
```

```
{
  int i;
  CHROMOSOME child;

  /* Argument Error Checking */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,
              "\nDC_GA_Crossover: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }

  child.fitness=0.0;
  child.gdp = (GENODATA *)malloc(sizeof(GENODATA));
  child.gdp->refcount = 0;
  child.gdp->genotype = (int *)malloc((DC_subproblem[pool].size+1)
                                      *sizeof(int));
  child.gdp->genotype[DC_subproblem[pool].size]='\0';
  child.gdp->id = chromo_id++;
  child.gdp->parent1 = parent1.gdp->id;
  child.gdp->parent2 = parent2.gdp->id;

  /* Do the crossover */
  for(i=0; i<DC_subproblem[pool].size; i++)
    if(rand_bit())
      child.gdp->genotype[i] = parent2.gdp->genotype[i];
    else
      child.gdp->genotype[i] = parent1.gdp->genotype[i];

  return(child);
}


/************************************************************/
/* No crossover; exists for the sake of investigating the null */
/* hypothesis in GA experiments.                            */
/************************************************************/

CHROMOSOME no_cross(CHROMOSOME parent1,CHROMOSOME parent2,int pool)
{
  CHROMOSOME child;

  child = parent1;
  return(child);
}
```

```
/* mutate.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Mutation.
 *
 * Modified by Luis F. Gonzalez to accept the new data structures defined
 * for the divide and conquer algorithm.
 *
 * Last Modified: Tue Sep  5 18:55:37 1995
 *
 */

/*
   Modification log:
   1. the mutation functions are added a new argument which is the
      pool number of the CHROMOSOME.
   2. scp_rows is substituted by the value DC_subproblem[pool].size
   3. define for minc initial value

*/

#include <stdio.h>
#include <stdlib.h>
#include "dcscpga.h"
#include "prototypes.h"
#include "errormsg.h"

#define MINIMUMCOST_INIT (1000000.0)

extern int scp_mutation_tournament;
extern int chromo_id;
extern double mute_rate;

extern double drandom();
extern long random();

extern char *malloc();
extern char *strncpy();

/*
   Testing code for mutation module.

int scp_mutation_tournament=2;
int chromo_id=2;
double mute_rate;

main()
{
  int genotype0[]={(int)'0',(int)'3',(int)'3'};
  int genotype1[]={(int)'5',(int)'4'};
  GENODATA chromodata={ genotype0,1,1,0,0 };
  CHROMOSOME initial,result;
  int npool,index,subindex,subsubindex;
  int costs[]={10,20,30,40,50,60};
  int colcov0[]={1,2,5};
  int colcov1[]={1,4,5};
  int colcov2[]={4,5};
  int colcov3[]={0,1,4,5};
```

```
int colcov4[]={0,1,2};
int colcov5[]={0,3};
int rowcov0[]={0,2};
int rowcov1[]={0,1,3};
int rowcov2[]={3,4};
int rowcov3[]={5};
int rowcov4[]={0,1,2,4};
SCP_COL col1={ 10,3,colcov0};
SCP_COL col2={ 20,3,colcov1};
SCP_COL col3={ 30,2,colcov2};
SCP_COL col4={ 40,4,colcov3};
SCP_COL col5={ 50,3,colcov4};
SCP_COL col6={ 60,2,colcov5};
SCP_ROW row0={2,rowcov0};
SCP_ROW row1={3,rowcov1};
SCP_ROW row2={2,rowcov2};
SCP_ROW row3={1,rowcov3};
SCP_ROW row4={4,rowcov4};

int pool0_row[]={0,1,2};
int pool1_row[]={3,4};

srandom((int)1234567890);
DC_problem.scp_rows=5;
DC_problem.scp_cols=6;
DC_problem.scp_costs=costs;

DC_col_coverage=(SCP_COL *)malloc(DC_problem.scp_cols*sizeof(SCP_COL));
DC_col_coverage[0]=col1;
DC_col_coverage[1]=col2;
DC_col_coverage[2]=col3;
DC_col_coverage[3]=col4;
DC_col_coverage[4]=col5;
DC_col_coverage[5]=col6;

DC_subproblem=(DC_ROW *)malloc(2*sizeof(DC_ROW));
DC_subproblem[0].size=3;
DC_subproblem[1].size=2;

DC_subproblem[0].rows=
  (int *)malloc(DC_subproblem[npool].size*sizeof(int));
DC_subproblem[0].dc_row_coverage=
  (SCP_ROW *)malloc(DC_subproblem[npool].size*sizeof(SCP_ROW *));
DC_subproblem[1].rows=
  (int *)malloc(DC_subproblem[npool].size*sizeof(int));
DC_subproblem[1].dc_row_coverage=
  (SCP_ROW *)malloc(DC_subproblem[npool].size*sizeof(SCP_ROW *));

DC_subproblem[0].rows=pool0_row;
DC_subproblem[1].rows=pool1_row;
DC_subproblem[0].dc_row_coverage[0]=row0;
DC_subproblem[0].dc_row_coverage[1]=row1;
DC_subproblem[0].dc_row_coverage[2]=row2;
DC_subproblem[1].dc_row_coverage[0]=row3;
DC_subproblem[1].dc_row_coverage[1]=row4;

printf("\nProblem Data:");
printf("\nSCP Rows: %d",DC_problem.scp_rows);
printf("\nSCP Cols: %d",DC_problem.scp_cols);
for(index=0;index<DC_problem.scp_cols;index++)
```

```
    {
      printf("\nCosts of Column %d:\t%d",index,
             DC_problem.scp_costs[index]);
      for(subindex=0;subindex<DC_col_coverage[index].ncovers;subindex++)
        printf("\nColumn %d\tcovers Row: %d",index,
               DC_col_coverage[index].covers[subindex]);
    }
  for(index=0;index<DC_problem.DC_nsubproblems;index++)
    {
      printf("\nPool: %d",index);
      for(subindex=0;subindex<DC_subproblem[index].size;subindex++)
        {
          printf("\nRow: %d\tNumber of Coverers: %d",
                 DC_subproblem[index].rows[subindex],
                 DC_subproblem[index].dc_row_coverage[subindex].ncoverers);

          for(subsubindex=0;
              subsubindex<DC_subproblem[index].dc_row_coverage[subindex]
              .ncoverers;subsubindex++)
            printf("\nCoverer #%d\tSet: %d",subsubindex,
         DC_subproblem[index].dc_row_coverage[subindex].coverers[subsubindex]);
        }
    }

  initial.gdp=&chromodata;
  initial.fitness=(double)0.5;
  initial.nactive=initial.ncover=0;

  for(index=0;index<10;index++)
    {
      printf("\nRun #%d",index);
      result=mutate_one_bit(initial,(double)0.5,0);
      printf("\nMutate One Bit GENOTYPE: ");
      for(subindex=0;subindex<DC_subproblem[0].size;subindex++)
        printf("%d ",result.gdp->genotype[subindex]-'0');

      result=mutate_multi(initial,(double)0.5,0);
      printf("\nMutate Multi GENOTYPE: ");
      for(subindex=0;subindex<DC_subproblem[0].size;subindex++)
        printf("%d ",result.gdp->genotype[subindex]-'0');
    }
  free(DC_subproblem[0].rows);
  free(DC_subproblem[1].rows);
  free(DC_col_coverage);
  free(DC_subproblem);
}
*/

/************************************************************************/
/* This flips a random 0/1 bit in 'original'.                              */
/* If given chromo has non-zero refcount, then work on copy instead.    */
/* Refcounts get adjusted upon insertion, later.                         */
/************************************************************************/

CHROMOSOME mutate_one_bit(CHROMOSOME original,double rate,int pool)
{
  int place,i, n_to_mutate;
  CHROMOSOME newchromo;

  /* Argument Error Checking */
```

```
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,
              "\nDC_GA_Mutation: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }

  if(rate>1||rate<=0)
    {
      fprintf(stderr,
              "\nDC_GA_Mutation: Attempt to use %d as mutation rate.\n"
              ,rate);
      exit(-1);
    }

  if(original.gdp->refcount > 0)
    {
      newchromo.gdp = (GENODATA *)malloc(sizeof(GENODATA));
      newchromo.gdp->refcount = 0;
      newchromo.gdp->genotype = (int *)malloc((DC_subproblem[pool].size+1)
                                        *sizeof(int));
      newchromo.gdp->genotype[DC_subproblem[pool].size]='\0';
      newchromo.gdp->id = chromo_id++;
      newchromo.gdp->parent1 = original.gdp->parent1;
      newchromo.gdp->parent2 = original.gdp->parent2;

      for(i=0;i<DC_subproblem[pool].size;i++)
        newchromo.gdp->genotype[i]=original.gdp->genotype[i];
    }
  else
    newchromo = original;

  /* choose a place */
 place = random()%DC_subproblem[pool].size;

 /*  n_to_mutate = (int)(mute_rate*(double)DC_subproblem[pool].size); */
  newchromo.gdp->genotype[place] = '0' +
    random()%DC_subproblem[pool].dc_row_coverage[place].ncoverers;

  return(newchromo);
}


/**********************************************************************/
/* This changes a random allele in the original, to lie in range '0' to */
/* '0'+max_allele. Forces a different value to to the current one.      */
/* If given chromo has non-zero refcount, then work on copy instead.    */
/* Refcounts get adjusted upon insertion, later.                        */
/**********************************************************************/
CHROMOSOME mutate_multi(CHROMOSOME original,double rate,int pool)
{
  int place,i, best;
  int ncoverer, coverer, cost;
  CHROMOSOME newchromo;
  double minc;

  /* Argument Error Checking. */
  if(pool>=DC_problem.DC_nsubproblems)
    {
```

```
      fprintf(stderr,
              "\nDC_GA_Mutation: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }
  if(rate>1||rate<=0)
    {
      fprintf(stderr,
              "\nDC_GA_Mutation: Attempt to use %d as mutation rate.\n"
              ,rate);
      exit(-1);
    }

  if(original.gdp->refcount > 0)
    {
      newchromo.gdp = (GENODATA *)malloc(sizeof(GENODATA));
      newchromo.gdp->refcount = 0;
      newchromo.gdp->genotype = (int *)malloc((DC_subproblem[pool].size+1)
                                              *sizeof(int));
      newchromo.gdp->genotype[DC_subproblem[pool].size]='\0';
      newchromo.gdp->id = chromo_id++;
      newchromo.gdp->parent1 = original.gdp->parent1;
      newchromo.gdp->parent2 = original.gdp->parent2;

      for(i=0;i<DC_subproblem[pool].size;i++)
        newchromo.gdp->genotype[i]=original.gdp->genotype[i];
    }
  else
    newchromo = original;

  for(place=0;place < DC_subproblem[pool].size; place++)
    if(drandom() <= rate)
      {
        /* Do scp mutation. Pitch scp_mutation_tournament against each */
        /* other, and take the best */
        minc=MINIMUMCOST_INIT;
        best=0;

        for(i=0;i<scp_mutation_tournament;i++)
          {
            ncoverer=(random()%DC_subproblem[pool].dc_row_coverage[place]
                      .ncoverers);
            coverer=DC_subproblem[pool].dc_row_coverage[place]
              .coverers[ncoverer];
            cost = (double)DC_problem.scp_costs[coverer]/
              (double)DC_col_coverage[coverer].ncovers;

            if(cost<minc)
              {
                minc=cost;
                best=ncoverer;
              }
          }
        ncoverer = '0' + best;
        newchromo.gdp->genotype[place] = ncoverer;
      }

  return(newchromo);
}
```

```
/*********************************************************************/
/* Function mrate() points at one of the next two functions, depending  */
/* on whether adaptive is chosen or note. Adaptive mutation scales the  */
/* probability of mutating each bit by the degree of similarity of the  */
/* parents.                                                             */
/*********************************************************************/

/*********************************************************************/
/* This function returns the bit mutation rate appropriate to the given */
/* parents - a number in the range 0..mute_rate, depending on degree of */
/* similarity of parents.                                               */
/*********************************************************************/

double adaptive(CHROMOSOME parent1, CHROMOSOME parent2, int pool)
{
  int loop,count;
  double prob,newrate;

  /* Argument Error Checking. */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,
              "\nDC_GA_Mutation: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }

  count=0;
  for(loop=0; loop<DC_subproblem[pool].size; loop+=1)
    {
      if (parent1.gdp->genotype[loop] == parent2.gdp->genotype[loop])
        count++;
    }
                        /* Proportion = observed / total. */
  prob=((double)count) / ((double)DC_subproblem[pool].size);

  newrate=prob*mute_rate;
  if(newrate>1)
    return((double)1);
  else
    return(prob*mute_rate);
}


/*********************************************************************/
/* This function simply returns the fixed mutation rate.                */
/*********************************************************************/

double nonadaptive(CHROMOSOME parent1, CHROMOSOME parent2, int pool)
{
  /* Argument Error Checking. */
  if(pool>=DC_problem.DC_nsubproblems)
    {
      fprintf(stderr,
              "\nDC_GA_Mutation: Attempt to access a non existing pool: %d.\n"
              ,pool);
      exit(-1);
    }
  return(mute_rate);
}
```

```
/* help.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 *
 * Help invoked by -h lag or by command-line foulup.
 *
 * Modified by Luis F. Gonzalez to include/adapt to the new DCSCPGA
 * arguments.
 *
 */

#include "dcscpga.h"
#include "errormsg.h"
#include "prototypes.h"
#include <stdio.h>
#include "version.h"

/**********************************************************************/
/* Help for the user                                                  */
/**********************************************************************/
void dcscpga_help(void)
{
fprintf(stderr,"\n\n%s, version %s\n",TITLE,VERSION);
fprintf(stderr,"Author: %s - %s\n",AUTHOR,DATE);
fprintf(stderr,"Based on: %s\n\n",PGA);
fprintf(stderr,"ATTENTION: ");
fprintf(stderr,"Options -F and -P, if used, must be called in that order.\n");
fprintf(stderr,"   -F<file> Use problem datafile <file> (scpdata).\n");
fprintf(stderr,"   -P<n>    Set number of Subproblem Pools. (5)\n");
fprintf(stderr,"   -p<n>    Set number of chromosomes per population. (50)\n");
fprintf(stderr,"   -NO<n>   Non-interactive, stop when One reaches <n>.\n");
fprintf(stderr,"   -No<n>   Like -NO<n> but also save final chromosomes.\n");
fprintf(stderr,"   -NA<n>   Non-interactive, stop when All reach <n>.\n");
fprintf(stderr,"   -Na<n>   Like -NA<n> but also save final chromosomes.\n");
fprintf(stderr,"   -l<n>    Set # of generations per stage. (1000)\n");
fprintf(stderr,"   -i<n>    Set the reporting interval. (50)\n");
fprintf(stderr,"   -S<n>    Seed the random number generator. (from clock)\n");
fprintf(stderr,"   -a       Adaptive mutation flag (FALSE)\n");
fprintf(stderr,"   -A       Pool Assignment (random)\n");
fprintf(stderr,"   -m<n>    Set mutation rate. (0.02)\n");
fprintf(stderr,"   -c<n>    Set crossover rate (Steady State). (0.6)\n");
fprintf(stderr,"   -b<n>    Set selection bias. (1.5)\n");
fprintf(stderr,"   -C<op>   Set crossover operator. (two)\n");
fprintf(stderr,"   -M<op>   Set mutation operator. (single)\n");
fprintf(stderr,"   -s<op>   Set selection operator. (tn3)\n");
fprintf(stderr,"   -r<op>   Set reproduction operator. (one)\n");
fprintf(stderr,"   -escp<t> Set mutation tournament for SCP. (5)\n");
fprintf(stderr,"      bds<t> Alternatively sets the mutation tournament\n");
fprintf(stderr,"            and uses Bus-Driver Evaluation.\n");
fprintf(stderr,"   -h       Display this information.\n");
fprintf(stderr,"   <file>   Also log output in <file>. (none)\n\n");

fprintf(stderr,"   Pool Assignment types ... random, seq.\n");
fprintf(stderr,"   Crossover operators ... one, two, uniform, none.\n");
fprintf(stderr,"   Mutation operators  ... single, multi\n");
fprintf(stderr,"   Selection operators ... random, rank, fitprop,\n");
fprintf(stderr,"                           tnK (K=integer > 1),\n");
fprintf(stderr,"                           tmK (K=integer > 1).\n");
fprintf(stderr,"   Reproduction operators ... one - steady state,\n");
```

```
fprintf(stderr,"                                    onecross - simple steady state\n"
        );

}
```

```
/* readdata.c
 * Copyright (C) 1993, 1994 Peter Ross and Geoff Ballinger.
 * This is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License, see the file COPYING.
 */

#define TRUE 1
#define FALSE 0

#include <stdlib.h>
#include <stdio.h>

/**************************************************************/
/* Get the next number from the input: put it in the location  */
/* addressed by second argument. This function returns 0 on    */
/* EOF. If stopateol is true, it returns -1 when it hits \n    */
/* (after which some other procedure has to read past the \n), */
/* otherwise it continues looking for the next number.         */
/* A number has an optional sign, perhaps followed by digits,  */
/* perhaps followed by a decimal point, perhaps followed by    */
/* more digits. There must be a digit somewhere for it to count*/
/* as a number. So it would read any of:                       */
/*  -.5                                                        */
/*  -0.5                                                       */
/*  -.5.7                                                      */
/* as minus-a-half. In the last case, it would read .7 next    */
/* time around.                                                */
/*   There doesn't seem to be a neat and reliable way to do    */
/* all this, including stopateol, using scanf?                 */
/**************************************************************/

double getdouble(file,valaddr,stopateol)
FILE * file;
double *valaddr;
int stopateol;
{
  int c;
  int found = FALSE, indecimal = FALSE;
  int sign = +1;
  double n = 0.0, f = 0.0, p = 1.0;

  /* First find what looks like start of a number - the first digit. */
  /* And note any sign and whether we just passed a decimal point.   */
  do {
    c = fgetc(file);
    if(c == EOF) return (0);
    else if(stopateol && c =='\n') return(-1);
    else if(c == '+' || c == '-') {
      sign = (c == '+')? +1 : -1;
      c = fgetc(file);
      if(c == EOF) return (0);
      else if(stopateol && c =='\n') return(-1);
    }
    if(c == '.') {
      indecimal = TRUE;
      c = fgetc(file);
      if(c == EOF) return (0);
      else if(stopateol && c =='\n') return(-1);
    }
    if(c >= '0' && c <= '9') {
```

```
      found = TRUE;
    } else {
      sign = +1;
      indecimal =  FALSE;
    }
  } while(!found);

  /* Now we've got digit(s) ... */
  do {
    n = 10.0*n + c - '0';
    p = 10.0*p;
    c = fgetc(file);

    if((c < '0') || (c > '9')) {
      found = FALSE;
      /* We've run out. If we already saw a decimal point, return now */
      if(indecimal) {
        if(c != EOF) ungetc(c,file);
        *valaddr =  sign * n/p;
        return(1);
      } else p = 1.0;
    }
  } while(found);

  /* We ran out and we didn't see a decimal point, so is this a decimal? */
  if(c != '.') {
    /* No, give it back to caller */
    if(c != EOF) ungetc(c,file);
    *valaddr = sign * n;
    return(1);
  } else {
    /* It is. Step past it, carry on hoping for more digits */
    c = fgetc(file);
    while(c >= '0' && c <= '9') {
      n = 10.0*n + c - '0';
      p = p*10.0;
      c =  fgetc(file);
    }
    /* We've run out of digits but we have a number to give */
    if(c != EOF) ungetc(c,file);
    *valaddr = sign * n/p;
    return(1);
  }
}

/* Use getdouble() above but convert result to int. */
int getint(f,valaddr,stopateol)
FILE *f;
int *valaddr;
int stopateol;
{
  int r;
  double x;
  r = getdouble(f,&x,stopateol);
  *valaddr = (int)x;
  return(r);
}

/* Use getdouble above but convert result to long. */
int getlong(f,valaddr,stopateol)
```

```
FILE *f;
long *valaddr;
int stopateol;
{
  int r;
  double x;
  r = getdouble(f,&x,stopateol);
  *valaddr = (long)x;
  return(r);
}
```