Generic Chromosome Representation and Evaluation for Genetic Algorithms

Kristian Guillaumier

Department of Computer Science and AI,
University of Malta

Abstract. The past thirty years have seen a rapid growth in the popularity and use of Genetic Algorithms for searching for optimal or near-optimal solutions to optimisation problems. One of the reasons for their immense success is the fact that the principles governing the algorithm are simple enough to be appreciated and understood. The major differences between one Genetic Algorithm and another lie within the schemes used to represent chromosomes, the semantics of the genetic operators, and the measures used to evaluate their fitness. Yet, these very differences make Genetic Algorithms so complex to design and implement when opposed with most real-world optimisation problems. The truth is that the people faced with these types of optimisation problems are not necessarily computer scientists or machine learning experts. Indeed, these types of problems constantly appear in various non-computing disciplines ranging from biology to manufacturing and economics.
In this report, we present a simple, yet powerful, high-level technique that can be used to describe the structure of chromosomes and how their fitness can be evaluated. The method is abstract enough to insulate the practitioner from all the implementation, design, and coding details usually associated with a Genetic Algorithm. Nonetheless, a wide array of optimisation problems ranging from the classical travelling salesman problem and the n-Queens problem to time-table scheduling and dynamic programs can be described.

1 Introduction – Setting the Scene

A basic Genetic Algorithm may be decomposed into the following steps:

- Create a starting population. Usually a set of random chromosomes are created.
- Repeat the following until some termination criterion is met:
  - Evaluate each chromosome using a fitness function.
  - Select pairs of chromosomes using some scheme such as random selection or fitness-biased methods.
  - Apply crossover on the pairs of chromosomes selected and mutation on individuals.
  - Create a new population by replacing a portion of the original population with the chromosomes 'produced' in the previous step.

The above algorithm may be implemented in any high-level programming language. However, in conventional implementations, most parameters, the fitness function, chromosome representation, and genetic operators are usually *hard-coded*. If the nature of the problem varies slightly or critical parameters change, the original code must be revised – sometimes substantially. Moreover, as already stated, the user may not be even computer literate and not prepared to deal with issues such as algorithm design, programming and debugging.

In this section we will present a simple lecture-timetabling problem and eventually show how the procedure can be made more abstract.

Suppose we wish to find the solution to a very simple time-timetabling problem where each of 5 lectures has to be assigned one of 3 timeslots. No lecture given by the same tutor may occur at the same time. Sample data may be found in the tables below.

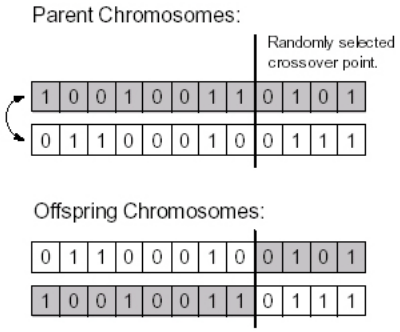
Lectures		
Code	**Name**	**Tutor**
1	CSA1	Andrew White
2	CSA2	John Green
3	CSA3	Michael Brown
4	CSA4	Mark Black
5	CSA5	Andrew White

Timeslots	
Code	**Description**
1	Monday 08:00
2	Monday 12:00
3	Monday 15:00

Our first task is to find a suitable chromosome representation. In such cases, a *direct representation scheme* may be used. We use a vector of symbols of length 5 (one per lecture) where the symbol at the ith position holds the timeslot assignment for the ith lecture. So, the following chromosome:

$$< 3, 1, 3, 2, 1 >$$

would be interpreted as the first lecture in timeslot 3, the second in timeslot 1, the third in timeslot 3, and so on. Once having found a suitable encoding scheme, we proceed by selecting our genetic operators. In order to keep things simple, we apply basic single point crossover and random mutation as shown in the following diagrams:

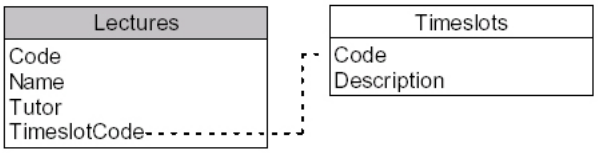


We finally decide on how chromosome fitness will be evaluated. In this case, the fitness function can be defined to return an integer value representing the number of times lectures given by the same tutor occur at the same time. Since the return value of the fitness function may be considered to be proportional to the severity of constraint violation, it may be interpreted as a penalty that has to be minimised. Once the chromosome representation, genetic operators, and the fitness function have been defined, our Genetic Algorithm can be implemented as outlined previously.

So far, we have considered a simple, yet perfectly realistic implementation of a Genetic Algorithm for lecture timetable-scheduling. However, it should be immediately apparent that once the nature of the problem changes, say by introducing rooms or soft-constraints, the whole program must be revised substantially. This is because all the effort has been hard-coded and hence cannot be easily extended.

2 Abstracting the Problem

Many optimisation problems, directly or indirectly depend on sets of static data (such as the Lectures and Timeslots tables) and certain relationships between them. In the lecture time-tabling problem we have seen earlier on, each row in the Lectures table must be associated with a, yet unknown, row in the Timeslots table. We call relationships that exist but are yet unknown, *dynamic relationships*. In view of this, we found it convenient to express such static data and relationships as a variation of database tables found in relational database management systems (RDBMSs). The following figure shows the Lecture and Timeslots tables, and the relationship between them.



Note: In the Lectures table, we have included a 4^{th} column called TimeslotCode. This column holds values matching those in the Code column of the Timeslots table and serves to create the relationship between the two. In database terminology, the TimeslotCode column is called a *foreign key column*.

These tables and their relationships represent the structure of the problem. Eventually this structure will determine the chromosome representation. The search process then attempts to find optimal, or near-optimal, values for the dynamic relationships – the TimeslotCode column above. Once the Genetic Algorithm starts producing candidate values for the dynamic relationships, a combination of queries, similar in concept to the *Structured Query Language (SQL) Select* statement and conditional *If-Then-Else* statements are used to evaluate fitness.

Suppose the Genetic Algorithm yields the following candidate values for the TimeslotCode dynamic column:

Lectures			
Code	**Name**	**Tutor**	**TimeslotCode**
1	CSA1	Andrew White	**3**
2	CSA2	John Green	**1**
3	CSA3	Michael Brown	**2**
4	CSA4	Mark Black	**1**
5	CSA5	Andrew White	**3**

Timeslots	
Code	**Description**
1	Monday 08:00
2	Monday 12:00
3	Monday 15:00

The following SQL statement may be used to retrieve the tutor and timeslot description from the two tables:

```
Select Lectures.Tutor, Timeslots.Description
From   Lectures, Timeslots
Where  Lectures.TimeslotCode = Timeslots.Code
```

Returning:

Lectures.Tutor	Timeslots.Description
Andrew White	Monday 15:00
John Green	Monday 08:00
Michael Brown	Monday 12:00
Mark Black	Monday 08:00
Andrew White	Monday 15:00

If we name the query above *qryTutorTimeSlots*, we can use the following conditional statement to return a penalty based on how many lectures are occupied during the same timeslot.

```
If TRUE Then
  Return {Number of tutors in same slot from qryTutorTimeslots} * 10000
Else
  Return 0;
```

Clearly, if no tutors are busy during the same timeslot, the if statement will return 0 – no penalty. Otherwise the value returned will be directly proportional the number of times the constraint has been violated.

3 Conclusion

In the previous section we briefly demonstrated how tables may be used to describe the structure of an optimisation problem, and how combinations of SQL-like queries and conditional statements can express the components of a fitness function.

This technique has been successfully implemented as a high-level modelling language called OPML[1] together with a general-purpose Genetic Algorithm-based runtime. A wide range of optimisation problems have been tackled including problems in linear programming, dynamic programming, lecture time-table scheduling, the travelling salesman problem, bin packing, and the n-Queens problem as test cases.

For a complete exposition the reader is referred to:

"A multi-purpose scripting language and interpreter for optimisation problems", Kristian Guillaumier, University of Malta, October 2002.

[1] Optimisation Problem Modelling Language