Snyman [24] for an algorithm that does not use the Hessian. Nocedal and Wright [15] review the augmented Lagrangian method which reduces the problems with ill-conditioning, but the theory required for this method is beyond the scope of this work.

Another method of dealing with constraints is to transform the problem so that the new independent variables can have any value. A possibility reviewed by Bandler [12] is

$$x_i = x_{li} + \frac{1}{\pi}(x_{ui} - x_{li}) \operatorname{arccot}(x_i') \tag{2.20}$$

where x_i is an independent variable of the original problem, x_{ui} and x_{li} are respectively the upper and lower bounds of the independent variable, and x_i' is the new independent variable. An unconstrained optimisation algorithm can now be applied to the new problem with $\mathbf{x}'$ as the independent variable. Nocedal [15] considers similar algorithms known as barrier function methods. The main problem with barrier function methods is that ill-conditioning of the Hessian can occur.

Other approaches using approximations of both the problem and the constraints in the neighbourhood of the current point have been developed. Nocedal and Wright [15] review the quadratic programming and sequential quadratic programming approaches. Snyman [36, 37] has recently proposed a new algorithm that uses a very simple approximation to the problems yet achieves excellent results. The main difficulty with these approaches is that a complicated sub-problem must be solved at each iteration, limiting the value of these algorithms where the evaluation of the objective function is fast. The main advantage of these approaches is that they typically require fewer objective function evaluations than the algorithms considered above making them very well suited to the case where the evaluation of the objective function is costly.

2.2 Genetic Algorithms

Genetic algorithms are numerical techniques that attempt to imitate evolution and natural selection. The motivation for this approach is the remarkable way in which natural systems

adapt to their environments. Genetic algorithms have been applied to a very wide range of problems including optimisation [17,33], computer programming [38], circuit design [38], the traveling salesman problem [17], and training of neural networks [39]. This document will focus on optimisation, but the conclusions are relevant to all applications of genetic algorithms.

There are some semantics surrounding the use of the term "genetic algorithm." The purists insist that only the binary genetic algorithms proposed by Holland [40] may be called genetic algorithms, with all other evolutionary techniques having different names. Michalewicz [33] uses the term "evolutionary program" to refer to any technique based on evolution and natural selection, but this can lead to confusion with "evolutionary programming," an algorithm for evolving finite state machines. For the purposes of this document the term "genetic algorithm" will be used to denote any technique based on evolution and natural selection, and the term "binary genetic algorithm" will be used to denote the form proposed by Holland [40].

Section 2.2.1 will consider genetic algorithms in more detail. Some well known selection schemes, data representations, and genetic operators will be presented to highlight the most important points. The Schema Theorem is discussed in Section 2.2.5 to show how genetic algorithms work. This result leads to some important observations concerning the implementation of genetic algorithms. Section 2.2.6 gives a brief introduction to messy genetic algorithms.

This is a very short introduction to genetic algorithms so only the most basic points are covered. The interested reader is referred to Whitley [41] for a genetic algorithms tutorial which briefly covers some advanced topics, Goldberg [17] for a detailed consideration of binary genetic algorithms, Michalewicz [33] for motivations for representations other than binary, and Bäck *et al.* [42] for a comparatively recent review of the field with over 200 references.

```
initialise population;
calculate fitnesses;
do
  {
    do
      {
        select two individuals;
        apply crossover with probability pc;
        apply mutation with probability pm;
      } until new generation is full;
    calculate fitnesses;
  } until termination criterion in met;
```

Figure 2.5: Pseudo-code implementation of a genetic algorithm.

2.2.1 Background

This section will review the most important characteristics of a genetic algorithm and two of the best known genetic algorithms will be presented as typical examples. The first is the binary genetic algorithm originally proposed by Holland [40] and covered in great detail by Goldberg [17], and the second is the real number genetic algorithm developed by Michalewicz [33].

As mentioned before, a genetic algorithm works by imitating evolution and natural selection as found in nature. The first step is to generate an initial population of individuals. Next a number of individuals are selected, favouring those with higher fitness, but not completely rejecting those with low fitness. These individuals are then allowed to breed, and some survive to the next generation. Mutation is also applied with a low probability and the whole process is then repeated until some termination criteria is met. The basic algorithm is shown in the pseudo-code implementation given in Figure 2.5.

From this discussion it is clear that five criteria have to be met for a genetic algorithm to be applied to a given problem. A representation of the data has to be established, an initial population of individuals must be created, a selection scheme must be chosen, genetic operators (crossover and mutation) need to be implemented, and a termination criteria has to be established.

The initial population is normally generated randomly, but this is not a requirement and individuals that are known to be good can be used to seed the initial population. The most commonly used termination criterion is to stop the algorithm after a fixed number of generations, but other possibilities do exist. One of these is to stop the algorithm when there has been no improvement in the best fitness for a number of generations. The representation of individuals, selection schemes, and genetic operators are considered in the following sections.

Section 2.2.2 reviews four of the most important selection schemes. The representation of the individuals processed by a genetic algorithm is considered in Section 2.2.3. Genetic operators are considered in Section 2.2.4.

2.2.2 Selection

Once a fitness measure for the problem being considered has been established, a way of selecting individuals must be implemented. This section will start with a description of the most important properties of a selection scheme, move on to address elitism, and will then consider four of the most common selection schemes in Sections 2.2.2.1 to 2.2.2.2. The information in this section is mainly obtained from Bäck [43], and Goldberg and Deb [44]

The selection scheme is what causes a genetic algorithm to converge, and thus has a dramatic effect on the performance of the algorithm. The genetic operators considered in Section 2.2.4 only create new individuals and have a comparatively small effect on the convergence of the algorithm. A selection scheme should favour the selection of good indi-

viduals to ensure that the algorithm converges to a good solution. However, poor individuals should still have a small chance of being selected to maintain diversity in the population and prevent premature convergence.

One of the problems with all the selection schemes described in this section is that they select individuals in a structured, but random manner. This means that there is usually no guarantee that the best individual from the current generation will survive to the next generation, so convergence is seldom monotonic. The simplest way to overcome this problem is to copy the best individual to the next generation, a process known as elitism [17]. Elitism is not required by $(\mu + \lambda)$ selection because the best individuals are guaranteed to survive to the next generation.

2.2.2.1 Description of Selection Schemes

Proportional selection is also known as Roulette wheel selection. Individuals are randomly selected with a probability equal to the ratio of the fitness of an individual to the sum of the entire population's fitnesses. This can be written as

$$p(\mathbf{x_i}) = \frac{f(\mathbf{x_i})}{\sum_{j=1}^{n} f(\mathbf{x_j})} \tag{2.21}$$

where $f(\mathbf{x_i})$ is the fitness of individual $\mathbf{x_i}$ and $p(\mathbf{x_i})$ is the probability of selecting individual $\mathbf{x_i}$. Proportional selection can obviously only be directly applied to problems where the fitness must be maximised and all possible fitnesses are strictly positive. However it is often possible to convert the fitness to this form and apply proportional selection to the modified fitness.

A subtle problem with proportional selection is that it becomes little more than random selection when a large number of individuals have similar fitnesses. This can happen near the end of a run when most of the population has converged to essentially the same fitness value. Another problem is that an individual that is much better than all others in a given generation will tend to get selected a large number of times, causing the population to

converge to that solution. This is a problem if an individual that is much better than all others, but which is not close to the best solution, is generated near the beginning of a run. Both of these problems can be overcome by scaling the fitness to be between some specified upper and lower bounds [17, 42]. Linear scaling is the most common form of fitness scaling, but nonlinear scaling can also be used. Nonlinear fitness scaling can be used to adjust the bias towards individuals with higher fitness [42]. The selection techniques listed below avoid these difficulties.

Selection probability is based on an individual's rank within a population in rank-based selection. Individuals that have a higher rank (better individuals) have a greater chance of being selected than individuals with a lower rank. The main advantages of rank-based selection are that it avoids problems with fitness scaling, and it can be used in cases where there is no absolute measure of fitness and individuals can only be compared. This is the case in game playing problems where the outcome of a match can only be a victory, loss, or draw. Linear ranking is the most common form of rank-based selection, but nonlinear ranking is also possible. As with fitness scaling, the bias towards better individuals can be adjusted in rank-based selection.

Tournament selection works by randomly choosing a number of individuals from the population to participate in a tournament. The individuals with the best fitnesses in the tournament are then chosen as the winners of the tournament, and are the results of the selection. This is usually implemented by randomly choosing two or three individuals at a time and then either selecting the individual with the best fitness or performing proportional selection on the individuals in the tournament. As with rank-based selection, tournament selection avoids fitness scaling problems and the bias towards better individuals can be adjusted. Tournament selection has the additional advantage that it is simple to implement.

The last selection methods considered here work by creating a population larger than that required and eliminating the worst individuals. In (μ, λ) selection the next generation is formed by selecting the best μ individuals from λ offspring, where μ is smaller than λ. The

process for $(\mu + \lambda)$ selection is similar except that the next generation is formed from the μ best individuals from the population consisting of both offspring and parents (μ individuals from $\mu + \lambda$ individuals). The main advantage of $(\mu + \lambda)$ selection over (μ, λ) selection is that $(\mu + \lambda)$ selection ensures that the best value improves monotonically. These two selection schemes have similar advantages to tournament selection, but they can have a stronger bias towards better individuals, and finer tuning of the bias can be achieved. These two schemes are common in algorithms based on Evolution Strategies [43].

2.2.2.2 Comparison

Detailed comparisons of these and other selection schemes can be found in the literature [17, 43, 44]. Goldberg and Deb [44] compare these selection schemes in terms of growth ratio, takeover time and time complexity, while Bäck [43] only considers takeover time.

Growth ratio is the ratio of the number of copies of an individual in the next generation to the number of copies of that individual in the current generation. The genetic operators (crossover and mutation) are ignored during a growth ratio calculation so that only the effect of the selection scheme used is seen. Proportional selection was found to have a growth ratio that is heavily dependent on the fitness function, and is high early on and low near the end of a run. Rank-based and tournament selection were found to have similar growth ratios that are much better than proportional selection. Genitor selection (a type of $(\mu + \lambda)$ selection) was found to have a growth ratio that is so high it actually causes problems.

Takeover time is defined as the time taken for the best individual to be copied to every position in a population but one. Again, the effect of crossover and mutation are ignored. Goldberg and Deb [44] found that the takeover times of all the selection schemes were of the same order of magnitude. The more detailed study conducted by Bäck [43] also considered the effect of changing the bias towards better individuals. Proportional selection was found to have a very high takeover time. Linear rank-based selection was shown to have a much

better takeover time than proportional selection, and tournament selection was better than linear rank-based selection. The shortest takeover times are obtained by (μ, λ) and $(\mu + \lambda)$ selection. The takeover times for rank-based, tournament, and (μ, λ) and $(\mu + \lambda)$ selection can be varied, with the largest range of variation being achieved by (μ, λ) and $(\mu + \lambda)$ selection followed by tournament selection.

The last parameter considered by Goldberg and Deb [44] is the time complexity (an indication of how much processing time is required) of each of these selection schemes. The most complex is proportional selection with a time complexity of order n^2, where n is the population size, followed by rank-based and Genitor (a type of $(\mu + \lambda)$) selection with time complexities of order $n \log(n)$. The simplest of the selection schemes considered here is tournament selection with a time complexity of order n. It is difficult to imagine a selection scheme with a time complexity of order less than n because n individuals must be selected.

2.2.3 Representation

The representation chosen for the individuals in a population can have a large effect on the performance of a genetic algorithm. Some general points will be considered here and two examples will be highlighted. Obviously different operators will be required for different representations.

The representation used should follow directly from the problem itself wherever possible. Important points to take note of are that the amount of data should be kept to a minimum, and features that are close together in the problem should be close together in the representation. An example of the first point is that a representation with 10 decimal digits accuracy shouldn't be used if only 3 decimal digits are required. The problems with having unnecessary data are that the algorithm takes longer to converge, and the chances that it will converge to a sub-optimum result are increased. An example of the principle that features that are close together in the problem should be close together in the representation

is that a two dimensional matrix would be a better representation for a two dimensional problem than a one dimensional vector. Both these points arise from the Schema Theorem presented in Section 2.2.5 and will be discussed further there.

The first genetic algorithms proposed by Holland used a binary string of the form

$$\mathbf{x} = x_{n-1}x_{n-2}\ldots x_3x_2x_1x_0 \tag{2.22}$$

where x_i are binary digits, and n is the length of the string. This representation can then be decoded to a single variable of n bits, two variables of $n/2$ bits, or m variables of n/m bits. The motivation for a binary representation comes from natural systems where genes are either present or absent. While this representation works well for integer variables, a large number of bits is required to encode real numbers which require very high accuracy or large ranges.

Michalewicz [33] overcomes this problem by using a number of floating point values to encode variables that are real numbers. In other words, each individual uses one floating point value to encode each system variable. In terms of (2.22), this means that each x_i is a real number and the problem has n variables.

2.2.4 Operators

After individuals have been selected, genetic operators are applied to create new individuals. The most important properties of genetic operators are presented below, followed by a consideration of mutation in Section 2.2.4.1 and crossover in Section 2.2.4.2.

Genetic operators are applied to modify current solutions in an attempt to produce improved solutions. As mentioned in Section 2.2.2, genetic operators have a comparatively small effect on the convergence of a genetic algorithm, but good genetic operators can lead to improved results.

Genetic operators are applied with a probability less than 1, so not all individuals are

affected, with unaffected individuals being copied to the new generation unaltered. This is necessary because genetic operators tend to have a disruptive effect as shown in Section 2.2.5.2. Allowing individuals to survive to the next generation without any modifications ensures that current solutions to the problem are retained.

Operators depend heavily on the representation used and fall into two general categories, mutation and crossover, with mutation being applied after crossover in most implementations. The binary operators originally proposed by Holland [40], and the floating point operators suggested by Michalewicz [33] will be presented below.

2.2.4.1 Mutation

Mutation operators take one individual as an input and randomly change that individual to ensure that genetic diversity is maintained in the population. This is necessary to ensure that the population does not prematurely converge to a poor solution. Mutation is a background operator that has a very small effect on the operation of most genetic algorithms, so it is applied with a very low probability. This does not mean that mutation in unnecessary, and a genetic algorithm that does not have mutation will not give good results. Binary, uniform, boundary, and non-uniform mutation will be presented as examples of typical mutation operators.

Binary mutation was originally suggested by Holland [40] as part of a binary genetic algorithm. Binary mutation works by randomly inverting a bit in the binary representation of an individual, which means that binary mutation can only be used with a binary representation. Binary mutation is usually applied with a probability such that one in a hundred to one in a thousand bits is affected [17].

Uniform mutation was introduced as part of a real number genetic algorithm by Michalewicz [33]. Uniform mutation modifies an individual by changing one of the variables in an individual to a random value uniformly distributed between the maximum and minimum allowable values of that variable.

Boundary mutation was introduced by Michalewicz [33] as part of a real number genetic algorithm to counteract the bias inherent in arithmetic crossover, and to effectively search the extreme values of each variable. Arithmetic crossover will be covered in detail below, but at this point it is sufficient to state that it tends to favour values near the middle of each variable's allowable range. Operators should not introduce any kind of bias because this will affect the operation of the genetic algorithm by favouring regions that do not necessarily produce good results. Boundary mutation randomly assigns either the maximum or minimum allowable value to one variable in an individual.

Non-uniform mutation was also introduced by Michalewicz [33] for use with a real number representation. This operator is similar to simulated annealing [14, 45] in the sense that the amount by which an individual is changed depends on the age of the population. Large changes are possible during the early part of a run when individuals are still far from good results, but only small changes are allowed near the end of a run when individuals are near good values. This is done to ensure that the search space is adequately covered near the beginning of the algorithm by allowing large changes, but good results are not lost near the end of the algorithm. One of the variables in a representation of the form shown in (2.22) will be changed according to

$$x_i' = \begin{cases} x_i + \Delta(t, u_i - x_i) & s = 0 \\ x_i - \Delta(t, x_i - l_i) & s = 1 \end{cases} \tag{2.23}$$

where x_i and x_i' are the old and new values of the variable respectively, l_i and u_i are the minimum and maximum values of the variable respectively, and s is a random bit that has an equal probability of being 0 or 1. The value of Δ used by Michalewicz [33] is given by

$$\Delta(t, y) = y\left[1 - r^{\left(1 - \frac{t}{T}\right)^b}\right] \tag{2.24}$$

where r is a random number uniformly distributed between 0 and 1, t and T are the current and final values of the population age (usually numbers of generations) respectively, and b is a parameter used to determine how much the mutation range changes with population age. Michalewicz [33] uses a value of 5 for b.

2.2.4.2 Crossover

Crossover is analogous to breeding in natural systems. The basic principle of crossover is that genetic information from two or more individuals is combined to form a new individual. Crossover is the most important operator in the vast majority of genetic algorithm implementations. This section will consider binary, simple, and arithmetic crossover.

Binary crossover was originally proposed by Holland [40] and is intended for use with a binary representation. Binary crossover works by selecting two parents and a cross point, and then copying the data from one parent before the cross point and from the other parent after the cross point. An example where two 8 bit parents are crossed after bit 5 to produce two offspring is shown below. The parents are given by

$$\begin{array}{ccc|ccccc} x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 \end{array} \tag{2.25}$$

and the offspring by

$$\begin{array}{ccc|ccccc} x_7 & x_6 & x_5 & y_4 & y_3 & y_2 & y_1 & y_0 \\ y_7 & y_6 & y_5 & x_4 & x_3 & x_2 & x_1 & x_0 \end{array} \tag{2.26}$$

where x_i and y_i are the bits of the parents, and | shows the cross point. In this case two offspring are generated for every crossover, but generating one offspring per crossover is also possible. Binary crossover is normally applied with a probability of around 0.6.

Simple crossover was introduced by Michalewicz [33] and is very similar to binary crossover, except that it is applied to representations with real numbers. The only difference to the case shown in the previous paragraph is that x_i and y_i are real numbers.

Simple crossover is very limited because it does not affect the values of the variables, so Michalewicz [33] also uses arithmetic crossover. This operator copies variables from one parent before the cross point, modifies the variable at the cross point using data from both parents, and then copies the remaining variables from the other parent. An example for the case where two parents with eight variables are crossed at variable 4 is shown below.

The parents are given by

$$\begin{array}{cccc|c|cccc} x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 \end{array} \tag{2.27}$$

and the offspring is given by

$$\begin{array}{ccc|c|cccc} x_7 & x_6 & x_5 & a_4 & y_3 & y_2 & y_1 & y_0 \end{array} \tag{2.28}$$

where x_i and y_i are real numbers, | shows the cross point, and a_i is given by

$$a_i = rx_i + (1 - r)y_i \tag{2.29}$$

where r is a random number between 0 and 1. It is also possible to generate two offspring from two parents as with binary crossover.

2.2.5 Schema Theorem

So far this work has concentrated on what genetic algorithms are and how to implement them, but nothing has been said about how and why genetic algorithms work. This is addressed by the Schema Theorem which explains the operation of genetic algorithms in terms of schemata (singular: schema). The Schema Theorem is covered by most books and tutorials that deal with genetic algorithms including [17, 33, 41]. While the Schema Theorem is only valid for binary genetic algorithms, the results can be applied to any genetic algorithm. This section will give a brief overview of the Schema Theorem to allow the reader to gain an understanding of how and why genetic algorithms work.

2.2.5.1 Schema Definition

The first step towards deriving the Schema Theorem is defining a schema. A schema is a template used to show similarities between different individuals. This is done by specifying the value of some of the bits in an individual, and leaving the values of all the other bits

unspecified. An asterisk is typically used as a "don't care" symbol to show which bits' values are unspecified. For example, all individuals which have the first bit of an eight bit string set to 1 are represented by the schema 1******* . Another example is that the strings 1010 1110 and 1110 1101 both contain the schema 1*1*****.

Schemata have two fundamental properties, order and length. The order of a schema, denoted by $o(H)$, is the number of bits whose values are specified (they are not "don't care" values). The schemata 1******* , ***0****, and ******1* all have order 1 because only one bit's value is specified. Examples of second order schemata are *1****0**, *00*****, and 1******* ***1. High order schemata match fewer strings than low order schemata, so the order of a schema is an indication of how general a schema is. The length of a schema, denoted by $\delta(H)$, is the number of bit positions from the first specified bit to the last specified bit. For example, the schema **1*0*0* has a length of 4 because the first specified bit is in position 1 and the last specified bit is in position 5, where the bits are numbered from right to left as shown in (2.22).

2.2.5.2 Derivation

The fundamental result of the Schema Theorem is a lower bound on the quantity of a given schema which will be present in a new generation. This value is affected by the number of copies of that schema in the current generation, the fitness of the schema relative to the average schema fitness, and the probability that crossover and mutation will disrupt the schema.

The first important step is thus to define the fitness of a schema, denoted by $f(H)$. A schema's fitness is the average of the fitnesses of all strings that contain that schema. For example, the fitness of the schema 1101 011* is the average of the fitnesses of the strings 1101 0110 and 1101 0111, and the fitness of *101 *001 is the average of the fitnesses of the strings 0101 0001, 1101 0001, 0101 1001, and 1101 1001. The fitness of a schema is implicitly processed by a genetic algorithm and need not be explicitly calculated. Schema fitness is

processed by fact that the individuals in a population which contain good schemata should have higher fitnesses than individuals that contain poor schemata. The accuracy of this assumption will improve as the population size increases because the number of copies of each schema will increase.

The number of copies of a given schema that are selected is now derived assuming that proportional selection (see Section 2.2.4.2) is used. Similar growth measure results for other selection schemes have been derived by Goldberg and Deb [44], but are not considered here because the results are similar. The probability that an individual, and thus a particular instance of a schema, will be selected is given by (2.21). The probability that any copy of a schema is selected is this probability multiplied by the number of copies of that schema in the current generation. This process is repeated until one individual has been selected for every space in the new generation. The number of copies of a schema that are selected for the next generation is thus given by

$$m(H, t+1) = m(H, t) n \frac{f(H)}{\sum_{j=1}^{n} f(\mathbf{x_j})} \tag{2.30}$$

where $m(H, t)$ is the number of copies of schema H in generation t, and n is the population size. This can be simplified by noting that the sum of all fitnesses divided by the population size is the average fitness of all individuals in the current population. So (2.30) becomes

$$m(H, t+1) = m(H, t) \frac{f(H)}{\bar{f}} \tag{2.31}$$

where $\bar{f}$ is the current population's average fitness.

The result in (2.31) shows that the number of good schemata (those with a fitness greater than the population average) will increase and the number of bad schemata (those with a fitness lower than the population average) will decrease. While this is desirable, it serves no purpose unless those schemata are processed to produce better results, and this requirement is met by the genetic operators described previously. Some schemata are destroyed and others are created during the application of genetic operators. When binary crossover takes place, schemata are destroyed when the cross point falls between bits whose values

are specified as shown below.

$$\begin{array}{lcccc|cccccc} \mathbf{x_i} & = & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ H_1 & = & 1 & * & * & 1 & * & * & * & 0 \\ H_2 & = & * & * & * & 1 & * & * & 0 & * \end{array} \tag{2.32}$$

In this example the cross point is chosen between bits 4 and 5. This is between two specified bits of schema H_1 so it will be destroyed unless the other individual taking part in the crossover has the same values in those bit positions. The cross point does not lie between any specified bits of schema H_2, so it is not affected by this crossover. Binary mutation will disrupt any schema that has a specified value that is mutated. Obviously new schemata that were not present in the individuals processed by the genetic operators will also be created.

The probability with which schemata are created and destroyed needs to be included in (2.31) to account for the effect of genetic operators. This analysis will only consider the probability that a schema will survive crossover and mutation, ignoring the creation of new schemata. This is a worst case analysis which establishes a lower bound on the probability that a schema will survive to the next generation.

A schema is disrupted by binary crossover if the cross point falls between the first and last specified bits of the schema. The cross point for binary crossover can be between any two bits, which means that an individual of length n has $n-1$ potential cross points. Every potential cross point has the same probability of being used, so the probability of using a specific cross point is $1/(n-1)$. The number of cross points between the first and last specified bits of a schema is equal to the length of the schema $\delta(H)$ so the probability of a schema being disrupted by binary crossover is thus $\delta(H)/(n-1)$. Binary crossover is not applied in every case, so this value must be multiplied by the probability that binary crossover is applied. The probability that a schema will survive crossover is given by

$$p_s = 1 - p_c \frac{\delta(H)}{n-1} \tag{2.33}$$

where p_c is the probability that crossover is applied. Equation (2.31) can now be modified

to account for the effect of crossover, giving

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{n-1}\right] . \tag{2.34}$$

Binary mutation can disrupt a schema by inverting any of the specified bits in a schema. The probability that any particular bit will be mutated is equal to the mutation probability p_m. The number of bits specified by a schema is given by the order of the schema $o(H)$ so the probability that a schema will be disrupted is given by $p_m o(H)$. This can be included in (2.34) to give the number of copies of a particular schema that can be expected in the next generation:

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{n-1} - p_m o(H)\right] . \tag{2.35}$$

As mentioned above, (2.35) ignores the fact that schemata are also created by crossover and mutation. This means that the value given above is actually a lower bound on the quantity of a schema that can be expected in the next generation, and thus represents the worst case. This can be shown explicitly by modifying (2.35) to use an inequality, giving

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{n-1} - p_m o(H)\right] . \tag{2.36}$$

2.2.5.3 Implications of the Schema Theorem

This section will consider some of the implications of the Schema Theorem and how they can be used to design better genetic algorithms.

The first important result is given in (2.31) and shows the expected number of copies of a schema in a new generation. The factor $f(H)/\bar{f}$ means that the number of copies of a given schema grows with the ratio of the fitness of the schema to the average population fitness. If the ratio of the schema fitness to the average population fitness is assumed to be a constant, (2.31) becomes a geometric progression. In other words, a schema with a fitness higher than the average fitness will get an exponentially increasing number of trials

assigned to it over time, assuming it is not disrupted by the genetic operators. A schema with fitness lower than the average fitness will get an exponentially decreasing number of trials assigned to it. This means that a population should very rapidly converge to a good result. Section 2.2.2.2 notes that proportional selection has the lowest growth ratio of the selection schemes considered, so the other selection schemes will do even better.

The final result given in (2.36) adds the effect of crossover and mutation to the result in (2.31). This equation shows that short schemata have a greater chance of surviving crossover than long schemata, and low order schemata have a higher probability of surviving mutation than high order schemata.

These conclusions lead to a result known as the building block hypothesis. A building block is simply a short, low order schema. The main result of the Schema Theorem is that good building blocks will be combined in such a way as to improve the population's fitness. This means that the representation chosen should be such that important parts of the problem are close together to minimise the possibility of being disrupted by crossover, and the amount of data required should be kept to a minimum to reduce the chances of disruption by mutation. Both these points were mentioned in Section 2.2.3.

The last issue that needs to be considered is the number of schemata that are processed by the algorithm per generation that contribute useful information about the problem. Not every schema that is present in the population will be usefully processed because long, high order schemata have a high probability of being disrupted by genetic operators. Goldberg [17] shows that the number of usefully processed schemata is of order n^3. So despite only explicitly processing n individuals per generation, a genetic algorithm implicitly processes on the order of n^3 schemata. This important result is known as implicit parallelism.

While the Schema Theorem is a very useful result which gives an insight into the operation of genetic algorithms, it is only valid for binary genetic algorithms and does not constitute a rigorous mathematical proof. The search for a generally applicable, mathematically rigorous proof of the convergence of genetic algorithms is ongoing. Leung *et al.* [46] give a brief

review of the most important results that have been achieved, and suggest a new model for analysing genetic algorithms.

2.2.6 Messy Genetic Algorithms

Genetic algorithms are a very powerful technique for solving difficult problems, but they are not perfect. One of the most difficult aspects of implementing a useful genetic algorithm is determining the representation. This is complicated by the fact that it is usually not possible to know in advance which ordering of variables will produce good results. Messy genetic algorithms [47,48] overcome this problem by modifying genetic algorithms to reduce ordering problems.

Section 2.2.6.1 will consider the motivation for messy genetic algorithms in more detail. The unique initialisation of messy genetic algorithms is presented in Section 2.2.6.2. Section 2.2.6.3 considers the representation used by messy genetic algorithms and highlights the difficulties this representation causes for fitness calculation. Section 2.2.6.4 considers the modifications that have to be made to conventional selection algorithms before they can be used for messy genetic algorithms. Section 2.2.6.5 presents the operators used in messy genetic algorithms. Lastly, Section 2.2.6.6 briefly considers the Schema Theorem as applied to messy genetic algorithms.

2.2.6.1 Motivation

The performance of a genetic algorithm can depend very strongly on the representation used. This is particularly evident in deceptive problems – problems that are known to be difficult for genetic algorithms. This section will give a brief overview of this difficulty.

The Schema Theorem states that short, low order, high fitness building blocks are combined to improve the population fitness. This means that features of a function must be tightly

linked (close together) in the representation so that the length of the important schemata are as short as possible. Unfortunately, it is not always possible to know in advance which ordering of the data will produce good results. This effect is accentuated when deceptive problems are considered. A number of techniques have been suggested for overcoming this problem, but none is satisfactory.

The first possibility is just to use a random ordering for every problem, but this is not a good solution because the probability of obtaining a good ordering is low. The other possibility is to introduce an operator which changes the ordering used in the representation during a run of the algorithm. This can be considered as a mutation operator for representation ordering. Goldberg *et al.* [47] give a number of difficulties with this approach. The most important objection to a reordering operator is that the genetic algorithm will then be trying to optimise both the ordering and fitness at the same time, producing a much more difficult problem than just improving fitness.

The problems highlighted above suggest that a new approach to genetic algorithm representation ordering is required. The messy genetic algorithm proposed by Goldberg *et al.* [47, 48] overcomes these ordering problems.

The inspiration for messy genetic algorithms, as with conventional genetic algorithms, comes from nature. Conventional genetic algorithms use the assumption that evolution takes place with a fixed number of genes which are either absent or present, but this is only true over comparatively short periods in evolution. Over longer periods of time this approximation is seen to be incorrect. In nature, individuals frequently have genes which are present more than once (overspecification), genes which are required but absent (underspecification), and chromosones which can vary in length. Messy genetic algorithms are unique because they allow varying length representations, underspecification, and overspecification.

2.2.6.2 Initialisation

The first major difference between messy genetic algorithms and conventional genetic algorithms is in the initialisation of the population.

The messy genetic algorithm starts with a large number of short individuals. These individuals are initialised to contain all possible building blocks of a specified, short length. The first phase of a messy genetic algorithm, known as the primordial phase, proceeds by only applying selection and gradually reducing the population size. This is necessary because a large number of individuals are produced during initialisation and it would be impractical to consider all these individuals in the complete algorithm. An additional advantage is that the use of selection means that mostly good individuals survive to the next phase, known as the juxtapositional phase, giving a very good starting point. In the juxtapositional phase both selection and genetic operators are applied.

2.2.6.3 Representation

The next important characteristic of any genetic algorithm is the representation used, and the unique aspects of the messy genetic algorithm representation are presented below.

In the case of messy genetic algorithms, the representation is of the form

$$\mathbf{x} = (n-1, x_{n-1})(n-2, x_{n-2}) \ldots (3, x_3)(2, x_2)(1, x_1)(0, x_0) \tag{2.37}$$

where the first number in each bracket is an index (i), and x_i is the value of bit i. The most important difference between this case and the binary genetic algorithm case given in (2.22) is the fact that each bit carries a label in this case. This is necessary because a messy genetic algorithm allows underspecification and overspecification in its representation. Another important difference is that the order of the bits in an individual is not important in a messy genetic algorithm. A typical individual is

$$(2,\ 1) \quad (4,\ 1) \quad (5,\ 0) \quad (4,\ 0) \quad (7,\ 1) \quad (1,\ 1) \quad . \tag{2.38}$$

Note that bits are not in order, bits 6, 3 and 0 are not present, and bit 4 is present more than once. The major challenge arising from this representation is finding a sensible way to calculate fitness.

There are two cases that cause difficulties with fitness calculation in a messy genetic algorithm: overspecification and underspecification. Overspecification is the simpler of the two cases to deal with, and Goldberg *et al.* [47, 48] simply use the first instance of any bit, so bit 4 would be 1 in the example in (2.38). The problem of underspecification is significantly more complex. Goldberg *et al.* [47] consider a number of possibilities and show that taking the missing bits from a locally optimal solution produces excellent results.

2.2.6.4 Selection

Once the fitness is calculated, the next step is to select individuals to create the next generation. Messy genetic algorithms can use the same selection algorithms as conventional genetic algorithms (see Section 2.2.2), but the fact that not all variables will be present in every individual does result in a few difficulties.

The main problem is to ensure that comparisons between individuals are valid. Comparing two individuals that do not have any variables in common is obviously not a good approach, but requiring too many variables to be common will limit the number of possible comparisons. Goldberg *et al.* [48] overcome this difficulty by only comparing individuals where the number of variables common to both individuals is greater than the number of variables that are expected to be common to both individuals due to the randomness inherent in the system. Goldberg *et al.* [48] have conducted a number of tests to prove the viability of this approach.

2.2.6.5 Genetic Operators

The last important characteristic of any genetic algorithm is the implementation of genetic operators. The messy genetic operators described by Goldberg *et al.* [47] are presented below.

The mutation operator is essentially the same as for the binary genetic algorithm with bits in the representation being randomly inverted with some low probability.

Crossover is also very similar to the binary genetic algorithm case except that the variable length of individuals must be accounted for. This is done by choosing the cross point for each parent independently. The offspring is then generated by copying data from one parent before its cross point and then from the other parent after its cross point. For example, if the parents are given by

$$\begin{array}{cccc|cc} (1,\,1) & (5,\,0) & (3,\,1) & & (6,\,1) & (1,\,0) \\ (5,\,0) & (6,\,1) & & & (3,\,0) & \end{array} \tag{2.39}$$

the offspring will be

$$(1,\,1) \quad (5,\,0) \quad (3,\,1) \;\big|\; (3,\,0) \tag{2.40}$$

where | shows the cross points. Note that the length of the offspring is different to that of its parents, and it is possible to produce more than one offspring.

2.2.6.6 Schema Theorem

Goldberg *et al.* [47] have extended the Schema Theorem to consider messy genetic algorithms. The most important differences from the form derived in Section 2.2.5 are that individual length is not a constant, and that variables can be masked if they are present more than once in an individual. The extension is significant because it shows that, despite the differences between messy genetic algorithms and conventional genetic algorithms their operation is similar. This is a very important result because the Schema Theorem was used to justify the development of messy genetic algorithms in the first place.