Chapter 6

Application of Genetic Algorithms to the Process Plan Optimisation Problem

6.1 Introduction

Genetic algorithms were successfully applied in a robust and general way to the process plan optimisation problem. This lead to a highly generalised extension into parallel plan optimisation as a new way of tackling the job-shop scheduling problem; this later work is described in the next chapter. Since these two aspects of the research described in this document are the most original, and its major contributions, in order that the reader may follow the technical material, the following section will give an introduction to genetic algorithms. An expanded version of this introduction can be found in [51]. The application of the technique to process plan optimisation is then described.
6.2 Genetic Algorithms

6.2.1 Introduction

Genetic Algorithms (GAs), originally developed by John Holland [49], are adaptive search strategies based on a highly abstract model of biological evolution. They can be used as an optimisation tool or as the basis of more general adaptive systems. The fundamental idea is as follows. A population of structures, representing candidate solutions to the problem at hand, is produced. Each member of the population is evaluated according to some fitness function. Fitness is equated with goodness of solution. Members of the population are selectively interbred in pairs to produce new candidate solutions. The fitter a member of the population the more likely it is to produce offspring. Genetic operators are used to facilitate the breeding; that is, operators which result in offspring inheriting properties from both parents (sexual reproduction). The offspring are evaluated and placed in the population, quite possibly replacing weaker members of the last generation. The process repeats to form the next generation. This form of selective breeding quickly results in those properties which promote greater fitness being transmitted throughout the population: better and better solutions appear. Normally some form of random mutation is also used to allow further variation. A simple form of this algorithm is illustrated in Figure 6.1. This population based survival of the fittest scheme has been shown to act as a powerful problem solving method over a wide range of complex domains [42, 43, 98, 6, 99, 23].

The analogies between GAs and natural evolution should be clear. The structures encode a solution to the problem, these are the genotypes. There will be some process for interpreting the structure as a solution: the phenotype. The interpretation is often implicitly embedded in the evaluation function and can be complex. When the encoding and the evaluation function are static, we are in the realms of optimisation. When they are not, the GA can be used to build adaptive systems; systems that are able to cope with a changing environment.
Create initial population of strings (genotypes)
... each string of symbols (genes) is a candidate solution to the problem

Assign a fitness value to each string in the population

Pick a pair of (parent) strings for breeding. The fitter the string the more likely it is to be picked

Put offspring produced in a temp_population

no

temp_population full?

yes

Replace population with temp_population

Figure 6.1: A simple genetic algorithm.
There are many different implementations of this idea, varying markedly in their specifics. Before giving the details of a commonly used model and then going on to explain why it works, it is worth stressing that the two most important elements to carry forward from this somewhat abstract description are selection and sexual reproduction.

6.2.2 Early Sequential Algorithms

The basic technique will now be described in some detail by referring to the early simple sequential algorithms. This is done as a matter of convenience and to give some historical perspective. Most of the underlying mechanisms are the same whether the implementation is sequential or parallel. It will become clear that the method is highly suitable for parallelisation and this topic is discussed later.

Holland’s early work was concerned with developing a powerful abstract general formalism for adaptive systems [47], this lead to his notion of a ‘general reproductive plan’ [48] which, slightly modified, was christened a genetic algorithm by Bagley [3]. All subsequent GA research has taken as its starting point the deceptively simple algorithm developed by Holland and his group at the University of Michigan during this period. The following subsections describe the technique’s essential elements.

The Population of Structures

The population of structures to undergo adaptation generally consists of strings (chromosomes) of a fixed length. Each element (gene) of the string represents some aspect of the solution and will have a set of possible values (alleles) mapped to various attributes. The fitness of such a string is measured by some objective function which costs the particular combination of attributes present. Hence the chromosomes may be, for instance, strings of real numbers, strings of integers, bit strings (string of 1s and 0s to be decoded into a set of parameter values), a permutation of some set of elements, a list of rules or some combination of these representations. Usually
each gene has a fairly small set of alleles. In fact there are theoretical arguments
[49, 37, 33] which suggest that very low cardinality alphabets should be used. This
goes a long way to explain why bit strings, with the lowest possible cardinality of two,
are so often employed. Indeed, a large part of the GA literature precludes discussion
of any other representation scheme. However, the analyses on which the result stands
are by no means complete. If a low cardinality alphabet string representation is
natural to the problem in hand, fine. If not, there is much empirical evidence that
high cardinality alphabet strings, especially when used with non-standard genetic
operators, can be effective [21, 24, 22, 58]. The process plan optimisation application
is described later in this chapter, the encodings used are fully elucidated.

Population size is an important parameter to consider when applying a GA to a
particular problem. If the population is too small premature convergence of the
whole population onto a local optima is very likely to occur. Populations sizes less
than fifty tend to cause difficulties. In practice, more complex problems require
larger populations. Figures of hundreds or thousands are not uncommon. Certain
parallel implementations require very large populations, as will be seen later. For
more detailed discussions of this topic see [35].

The Genetic Operators

The set of genetic operators developed by Holland, and the one generally used,
consists of three operators: crossover, inversion and mutation. Simple crossover
involves choosing at random a crossover point (some position along the string) for
two mating chromosomes, then two new strings are created by swapping over the
sections lying after the crossover point. Inversion is simply a matter of reversing a
randomly chosen section of a single string. Mutation changes the value of a gene to
some other possible value. The genetic operators are applied at the breeding stage
according to a routine like the following. When two strings are selected for breeding,
first apply crossover (with some high probability) and randomly choose one of the
two new strings thus formed. Next apply inversion (with a medium probability) to
this string. Each gene on the resulting string undergoes mutation (with a very low probability) and the outcome is taken as the offspring. The basic operators and the breeding process are illustrated in Figure 6.2. Note the stochastic nature of this process. All operators are applied probabilistically and crossover and inversion points are chosen randomly.

The overall effect is to emphasise combinations of basic building blocks (groups of genes) which produce maximum fitness.

In some problem domains it may be beneficial to allow dynamic length strings. This can be achieved by, for instance, randomly selecting different crossover points on each parent rather than forcing them to be the same. Other operators, such as translocation (moving a section of the string to a new location), may also be useful.

Action of The Operators

Crossover is generally considered the most important of the operators and serves two complimentary search functions. It provides new points for further testing within the hyperplanes, defined by gene combinations, already present in the population. It also introduces instances of new gene combinations by exchanging material between strings. The power of this operator will be analysed in the next subsection.

Alternative crossover operators such as two point crossover and uniform crossover (each gene is taken from the two parents with respective probabilities $p$ and $1 - p$ [1, 104]) are often used.

Inversion cannot always be easily used as it may produce illegal structures when applied in combination with simple crossover. This is particularly likely if the genetic coding is order sensitive. The purpose of this operator is to increase the linkage of gene combinations exhibiting good performance. It does this by reducing their length and hence making their disruption by crossover less likely. It can also bring previously widely separated alleles into close proximity. Holland has shown that its action is, like that of crossover, intrinsically parallel (see later).
1) apply crossover with high probability

parent1

|-------------------------------|

|-------------------------------|

parent2

randomly chosen crossover point

swap

randomly chosen inversion points

reverse

2) choose one of the resulting offspring at random, apply inversion with medium probability

3) there is a small chance of any gene undergoing a random mutation, resulting string is final offspring

offspring

mutation here

Figure 6.2: Application of the genetic operators.
**Mutation** is often considered a background operator, used to guarantee that the probability of searching a particular subspace of the problem space is never zero. It provides the crossover operator with the full range of alleles which should prevent the algorithm from becoming trapped on local optima. The probability of any gene undergoing mutation must be low enough to ensure that this process does not become overactive and start disrupting the advances made by crossover. Typically all genes on the string have equal but independent probabilities of mutation. In some problem domains relatively high mutation rates have been found to be useful. In these cases the mutation operator, acting in tandem with crossover, is taking a more active search role [53].

**Selection Mechanisms**

Selection, whereby more credence is given to fitter population members, provides the dynamo that powers the algorithm. However, this survival of the fittest scheme is more subtle than it at first seems. In this context survival of the fittest does not equate to the fittest will survive, but the fittest are *more likely* to survive. In more detail, the fittest are more likely to pass on some of their genes to later generations. This probabilistic element - which is found in other parts of the method, for instance the genetic operator mechanisms - helps to account for the technique’s power and robustness.

Four selection mechanisms are dealt with here:

- Breeding Pool
- Roulette Wheel
- Ranking
- Local rules

Each of these usually requires an explicit measure of fitness. In optimisation applications, the fitness function is directly related to the objective function of the
problem. Fitness is most often taken to be a non-negative merit value. If the problem is to maximise some objective function \( g(x) \), fitness may well just be taken as the value of \( g(x) \). More commonly optimisation problems involve the minimisation of the objective function. In this case the fitness function, \( f(x) \), is very often taken as:

\[
f(x) = \begin{cases} 
Cost_{\text{max}} - g(x) & \text{if } g(x) < Cost_{\text{max}}, \\
0 & \text{otherwise.}
\end{cases}
\]  

(6.1)

\( Cost_{\text{max}} \) is usually either the highest value of \( g \) observed so far, the highest value of \( g \) in the current population, or the highest value observed over the last \( k \) generations. Sometimes it relates to more complex population statistics.

**Breeding pool** selection works as follows. The fitness of each member of the current population is calculated. The relative fitness of each member is then calculated as follows:

\[
relf_i = \frac{f_i}{\sum_{i=0}^{N} f_i}
\]

(6.2)

where \( f_i \) is the fitness of the \( i \)th member of the population and \( N \) is the population size. The expected number of offspring for each individual is then calculated using Equation 6.3, \( \text{round} \) rounds to the nearest integer.

\[
\text{num\_offspring}_i = \text{round}(relf_i \times N)
\]

(6.3)

This number typically varies between zero and four or five. Each member of the population then has the appropriate \( \text{num\_offspring}_i \) copies of itself put into a temporary population, or breeding pool. Pairs for mating are then chosen at random from this pool. The offspring produced are used to replace the current population to form the next generation. Clearly the greater the number of copies of an individual in the pool, the more chance it has of contributing towards the next generation. Note some individuals, those with \( \text{num\_offspring}_i = 0 \), are excluded from breeding.
On each cycle the whole population might be replaced or some predefined proportion of it. In the latter case, it is usual to use a stochastic process which biases replacement towards the weaker members of the current population. An intermediate scheme, the so called elitist strategy is often practiced. In this scheme the whole population is replaced each generation except the fittest individual, which is carried through unscathed.

**Roulette wheel** selection is slightly different. Again, each member of the population has their relative fitness, $rel_f$, calculated. This value can be thought of as assigning the correct sized slice of pie, or an appropriately sized sector of a roulette wheel. Individuals are chosen for mating by generating a random number between 0 and 1 and then moving through the population an individual at a time until the cumulative relative fitness is greater than the random value. This is directly analogous to spinning the roulette wheel; the individual associated with the sector that comes to a halt opposite the pointer is selected. The bigger the relative fitness the more likely the individual is to be selected for breeding. Note that with this scheme no member of the population is excluded from breeding, they all have some chance of contributing to the next generation.

**Ranking schemes** for selection are particularly straightforward. Using this strategy the population is ranked, or ordered, according to the fitness values of its members. Selection is then performed by following a pre-determined probability distribution function, such as the ones shown in Figure 6.3. This may be a simple linear function that constrains the first ranked (fittest) individual to be twice as likely to be selected as the median ranked individual. This scheme tightly controls the selective pressure and allows strong differentiation of the population, even at later stages when their fitness values are very close. It has been argued that this can help to prevent premature convergence. Indeed, Whitley has developed a sequential variant of the genetic algorithm that incorporates ranked-based selection in a *steady state* algorithm [112]. That is, offspring are introduced into the population one at a time, rather than whole population at a time, which allows a more gradual, and supposedly more robust, search. Whitley’s algorithm can be summarised as follows:
1. Randomly generate population

2. Rank the population by fitness

3. Pick pair of parents by using a pre-determined ranked-based distribution

4. Breed to produce offspring

5. Insert in correct position in population (no repeats), push off bottom ranked member of population

6. Unless stopping criteria met, go to 3.

These selection schemes are intended for use with sequential GAs. An alternative form of selection, that only really makes sense in the context of distributed GAs, is governed by local rules of interaction. Briefly, the idea is that a population is somehow split up into many subpopulations, either explicitly or implicitly, and selection occurs *locally*. That is, with reference only to the subpopulation, not to the global population. Local schemes may be based on the methods described earlier or may be simpler. For instance, the population may be effectively spread out over a 2D grid and selection occurs among groups of very close neighbours. In this scheme, any individual has some small number of potential mates. One is chosen according to a probability distribution based on their fitnesses. The advantages of these sorts of schemes, and of distributed GAs in general, are discussed later.

Before closing this section there should be some discussion of fitness scaling. Relative fitness based selection, whether used in sequential or parallel GAs, introduces two problems, particularly when small populations are used. First, if a few good individuals appear early on in a run, relative fitness selection rules can easily allow an almost complete take over by these individuals, leading to premature convergence. Second, later in a run, although there may still be significant diversity in the population, the average fitness may be close to the best fitness. In these circumstances, relative fitness selection is not sensitive enough to push the search along. In other words, selective pressure has disappeared. Rank-based selection is one solution to
Figure 6.3: Rank-based selection probability distribution functions.
these problem, fitness scaling is another. Linear scaling is often used, the scaled fitness, $f'$, is related to the raw fitness, $f$, as follows:

$$f' = af + b$$

(6.4)

Where the coefficients $a$ and $b$ are chosen such that the average scaled fitness equals the average raw fitness. A more detailed discussion of these matters can be found in [37].

All of the details of GAS introduced above will be recognised, often in more advanced forms, in the genetic algorithm work described later.

**Schemata Analysis, Implicit Parallelism**

So far there have only been informal arguments to explain why GAs work: selection identifies useful building blocks and the genetic operators slot them together in more and more powerful combinations. This section provides a brief overview of Holland’s original formal analytic work which goes some way to explaining the power of GAs and gives further insight into the mechanics of the technique.

If a search strategy is to test for and incorporate structural properties associated with better performance, there must be some method of identifying these useful properties. This is the function of schemata; providing a means of comparing the properties of population strings. A schema describes a subset of strings with similarities at certain positions. Suppose we have a population of strings of length 6 where for each string position, $i$, there are three possible values \{a, b, c\}. If we add the symbol * to mean don’t care - i.e. it is irrelevant which value is taken at that point, we can represent schemata. The schema $a_1a_2**b_5c_6$ matches 9 strings, those with the fixed values shown in the 1st, 2nd, 5th and 6th positions, but any of the three legal values in both the 3rd and 4th positions. In this example, with an alphabet of cardinality 3 for each string position, the number of possible schemata is $4^6 = 4096$. For a population of $N$ strings of length $l$, there will be an upper bound
of \( N2^l \) different schemata present. \( 2^l \) is the number of schemata contained in any string - each position may take on its actual value or the * symbol. It can be seen that contained within these schemata there is a very large amount of information about the similarities between strings in a given population.

During the reproduction phase the fittest strings provide the most copies for the breeding pool, it follows that schemata associated with high fitness are sampled more often than those that are not. To understand the effect of crossover we need another simple definition. The distance between the first and last specified (fixed value) string positions in a schema is known as the schema's defining-length. For instance, the schema \( b_1 * c_2 a_4 * \) has defining-length \( 4-1=3 \) and \( a_1 a_2 * * * \) has defining-length \( 2-1=1 \). Clearly, schema with short defining-lengths are less likely to be disrupted by crossover than those with long defining-lengths. It can be seen intuitively, then, that the combined effect of selection and crossover is to propagate highly fit short defining-length schemata throughout the population. These schemata are generally known as 'building blocks'. It is these building blocks which combine to produce fitter and fitter strings.

To put things on a firmer basis, Holland has shown [49] that, under certain assumptions, a GA gives exponentially increasing sampling to the observed best schemata. This is an extremely important result and goes a long way to explaining the power of the method. The result Holland gave, for a GA using crossover and mutation (a commonly used algorithm), was:

\[
P(S, t + 1) \geq \left[ 1 - P_c \frac{\delta(S)}{l - 1} (1 - P(S, t)) \right] (1 - P_m)^{\text{O}(S)} \frac{f(S)}{\bar{f}} P(S, T)
\]

(6.5)

Where, \( P(S, t) \) is the expected proportion of schema \( S \) in the population at generation \( t \), \( P_c \) and \( P_m \) are the probabilities of crossover and mutation, \( l \) is the population string length, \( \delta(S) \) is the defining length of \( S \), \( f(S) \) is the fitness of \( S \), \( \bar{f} \) is the average schema fitness, \( \text{O}(S) \) is the order of \( S \) number of defined (fixed value) positions in

\footnote{The assumptions mean that the result is not all embracing, or even particularly rigorous, but the flavour of the result has been observed to hold time after time.}

108
Assuming $P_m$ is very small (see previous section), and ignoring negligible cross products, this can be rewritten as:

$$P(S, t + 1) \geq P(S, t) \frac{f(S)}{\bar{f}} \left[ 1 - \frac{P_c \delta(S)}{t-1} - O(S)P_m \right]$$ (6.6)

Although Holland's derivation of Equation 6.5 is not particularly difficult to follow, it is too long to reproduce here. However, by examining Equation 6.6, we can see how the result implies exponentially increasing sampling for fit schemata. If we assume $S$ is a constant amount fitter than average, i.e. $f(S) = C\bar{f}, C > 1$, then Equation 6.6 can be rewritten as Equation 6.7, where $K$ is a constant greater than 1.

$$P(S, t + 1) \geq P(S, t)K$$ (6.7)

Tracing this recursive relation back to generation 0, we readily obtain Equation 6.8.

$$P(S, t) \geq P(S, 0)K^t$$ (6.8)

That is, fitter than average schemata receive exponentially increasing sampling as the GA runs from generation to generation. Similarly, schema less fit than average ($f(S)/\bar{f} < 1$) will receive exponentially decreasing sampling. It follows from Equation 6.6 that $K$ will be larger for low defining-length ($\delta(S)$) schemata. This supports the building blocks explanation of the efficacy of GAs. Note that tracing the argument back to generation 0 makes many assumption about the intervening population dynamics controlling relative schemata fitnesses. As mentioned earlier, these results are not strictly rigorous (as is often assumed and stated) but they are the best we have and appear to hold a large portion of the true story.

In the same work Holland also showed that in a population of $N$ strings, although...
a GA processes $N$ strings per generation, it processes in the order of $N^3$ schemata. He refers to this equally important result as the *implicit parallelism* of genetic algorithms.

These two fundamental results follow from the very straightforward mechanics of a GA, they do not require any special accounting procedures or complicated transformations, just a population of strings and the application of the simple operators described.

**Other Analyses**

The formal analysis of genetic algorithms is an active area of research. Most of the research is aimed at understanding under what circumstances a GA will perform well and when it will perform badly. There are a number of interrelated questions on how to encode the problem, how to set the algorithm control parameters, such as population size and genetic operator probabilities, and how to control selection. A discussion of this work is outside the scope of this chapter. However, it is worth noting that most analyses take Holland’s schema work as their starting point. The non-linear dynamics of a typical GA system makes more detailed and more rigorous analysis very hard. This, coupled with the diversity of application details, means that empirical results tend to hold more weight. Some references on GA analysis worth pursuing are [33, 34, 36, 71] [93, 94].

**Advantages of GAs**

Because Holland and his students developed genetic algorithms to serve as adaptive problem solving strategies able to operate over a large range of environments, they have qualities that make them suitable for many large combinatorial problems and string representable search tasks. By a combination of selection and reproduction via genetic operators, they are able to find very fit structures by searching only a tiny proportion of the whole problem space. As long as the string representations
and the cost function are accurate, GAs can conduct a successful search without recourse to any special domain specific heuristics. The subtlety of their action tends to prevent them from getting stuck on local optima and ensures that they simultaneously search widely separated parts of the problem space. This is largely due to the random elements in the action of the genetic operators and the fact that they search from a population of solutions rather than just one. No assumptions need to be made about the search space, often in contrast to the situation with branch and bound and various heuristic search techniques. These qualities make genetic algorithms an extremely robust problem solving method. It is this robustness that makes them an attractive and useful search method. GAs work by combining candidate solutions together to produce new candidate solutions. They identify building blocks which promote greater fitness and propagate these through the population producing better and better solution strings. Of course this implies that the genotypic encoding describes a search space that is amenable to this process: that contains fairly independent building blocks that can be combined in useful ways. This requires that an otherwise random string with one good building block is fitter than a random string with no good building blocks and that the building blocks do not interact in such complex ways that they cancel out each others fitness over significant parts of the search space.

In general, GAs do not provide a panacea. A great deal of ingenuity is often needed to derive a suitable encoding and provide it with an appropriate set of genetic operators. However, when this is achieved, especially in fields where existing techniques are weak, GAs can be a remarkably powerful tool.

6.3 Applications

By now there are hundreds of successful applications of GAs. A few will be mentioned here, but consult the following for many more examples [42, 43, 98, 6, 22, 23].

GAs have been widely used in the optimisation of multimodal continuous functions
They have advantages over traditional gradient descent techniques in terms of robustness and avoidance of local optima. Typically, a complex function of several variables, $f(x_1, x_2, ..., x_n)$, is to be minimised. The genotype used is a simple bit string, 1001001010...001, where each gene can take the value 0 or 1. The string is decoded in chunks, where each chunk is taken as the binary value of one of the parameters $x_1...x_n$. Fitness is measured in terms of the value of $f$, as described earlier in this chapter. Much of the early empirical evidence for the usefulness of GAs comes from De Jong's 1975 study of their performance as function optimisers [25]. Non-function optimisation applications include pipeline and structure optimisation [38], machine design [64], and systems identification [63].

GAs have been applied to such practical combinatorial optimisation problems as production plan optimisation [57, 110], scheduling [21, 113, 16, 58] [52, 82], bin packing [100], compaction of symbolic layout [31], circuit partitioning [50], and timetabling [18, 72].

An application, falling between function optimisation and combinatorial optimisation, that has stirred up a lot of interest is the optimisation of component designs. Powell et al. [92] have developed a domain independent design optimisation tool that integrates expert systems and genetic algorithms. Bramlett and Cusic have applied a related technique to the parametric design of aircraft [10].

To close this section it is worth mentioning two application principles that have been found useful: one, if there are domain search algorithms available consider hybridizing them with a GA, two, if there are useful domain heuristics incorporate these into a GA by developing appropriate encodings and recombination operators.

### 6.4 Parallel and Distributed Genetic Algorithms

An extended form of parallel distributed genetic algorithm is developed in the next chapter, so it is appropriate to close this section on genetic algorithms with a brief look at parallel versions of the algorithm. It should be clear from the preceding
sections that the genetic algorithm is highly parallel in form. From the very early
days of its development its potential for parallelisation, with all the attendant ben-
efits of efficiency, have been noted [47]. Mainly because of the lack of availability of
hardware, it is only recently that significant work has been done in this direction.
The more successful parallel implementations are more than just simple translations
of the sequential algorithms into a multi-processor environment. After all, the early
algorithms were sequential because of hardware limitations, they were inevitably
impoverished in the way they handled population-distributed processing and inform-
ation.

In the early 1980s Grefenstette studied four proposed parallel implementations [40].
These were:

- Synchronous master-slave
- Semi-synchronous master-slave
- Distributed, asynchronous concurrent
- Network

In the first of these implementations, the master processor controls selection and
mating while the slave processors do the evaluations of population members. This
makes some sense given that evaluation is very often the bottleneck in the sequential
algorithm. However, it is rather wasteful if there is much difference in evaluation
times and it relies on the health of the master processor. It is suitable for imple-
mentation on SIMD machines and could potentially allow the use of very large pop-
ulations. The second model overcomes the first of the drawbacks encountered with
the synchronous model by relaxing the strict synchronous requirement. The third
model is fully asynchronous and concurrent; each processor performs selection, mat-
ing and evaluations independently. The processors access a common shared memory
storing population information. There is a requirement that population members
cannot be simultaneously processed by different processors. This implementation is
highly robust, if not as subtle as some of the later versions described below. The
final, network, model involves a number of independent simple GAs each with its own memory. Every so often the best individuals discovered by each population are broadcast to the other populations. This requires far less communication than the other schemes and is suitable for MIMD or LAN architectures. It was some time before results from real implementations of these models appeared [91].

Cohoon et al. [62] presented a model inspired by the theory of punctuated equilibria [28]. Cohoon's algorithm is similar to Grefenstette's fourth model but with some refinements. Their model involves \( N \) subpopulations of size \( n \), each re-iding on an individual processor. Each processor executes a sequential genetic algorithm for some fixed number of generations (an epoch). At the end of an epoch the population will be approaching equilibrium, but will still have some diversity. Next, each subpopulation copies a random subset of itself to neighbouring subpopulations. Each processor now has a surplus of solutions and so makes a probabilistic selection of \( n \) of them to serve as its initial population for the next epoch. The cycle continues. This process mimics the stasis-catastrophe-rapid evolution-stasis cycle of Eldrege and Gould's theory. Cohoon applied this algorithm to the Optimal Linear Arrangement problem and reported:

In our experiments, the result was more than just a hardware acceleration, rather better solutions were found with less total work. (Cohoon et al. 1987).

By 1989 more sophisticated parallel GAs had started to appear. Muhlenbein and Gorges-Schleuter developed their parallel asynchronous ASPAROGOS algorithm [77, 76, 39] which employs a number of interesting extensions derived from population genetics theory. Individuals live on a 2D grid with local selection operating. An important addition is that when each individual comes into existence it does local hill climbing to improve its fitness. Experiments were performed on producing offspring using polysexual voting recombination. The basic algorithm is represented below, it is applied asynchronously and in parallel to all individuals.
1. Randomly generate $N$ individuals

2. Each individual does local hill climbing to increase fitness

3. Each individual chooses partner(s) for mating in its neighbourhood, according to their rank in the neighbourhood.

4. Create new offspring with genetic operators

5. Replace individual

6. If not finished, go to 2.

In their neighbourhood model there is a notion of subpopulations ‘isolated by distance’, but neighbourhoods do overlap. The population is viewed as a continuous structure, with local interactions only, making it quite different from the isolated subpopulation models. Manderick and Spiessens have reported a number of experiments (Manderick & Spiessens 1989; Manderick & Spiessens 1991) with a fine grained parallel genetic algorithm implemented on massively parallel machines such as the Connection Machine and the DAP. Like Muhlenbein and Gorges-Schleuter, they use a population distributed over a planar grid with selection restricted to small neighbourhoods on the grid. Their neighbourhoods are larger than Muhlenbein’s and their selection mechanisms are different (simple relative fitness, rather than ranking).

More recent work has refined some of these models and explored the application of parallel GAs to highly complex problems requiring very large populations. Collins and Jefferson have explored a number of selection mechanisms in sequential and parallel GAs [17]. The results of their studies confirmed suspicions welling up in the GA community. Although they were careful to point out that further studies should be carried out, they concluded that local selection and mating was superior to older schemes over a very broad range of problems. More specifically, local selection and mating

1. finds optimal solutions faster (in terms of solutions evaluated);
2. typically finds multiple optimal solutions in the same run;

3. is much more robust than global schemes.

On top of these, local selection and mating schemes produce more efficient parallel implementations by cutting down on interprocessor communications and by largely eliminating sequential bottle necks, such as relative fitness calculations. Similar conclusions have been drawn by Davidor [20].

It should be stressed that although distributed GAs (i.e. GAs with geographically spread population and local selection operating) are highly suitable for parallelisation, they retain all their other advantages when implemented on a single processor.

The parallel distributed algorithms presented in the next chapter have similarities to some of the above and were developed, independently, at about the same time [58, 52, 51].
6.5 Application of Genetic Algorithms to Process Plan Optimisation

6.5.1 Introduction

The initial investigation into employing GAs to tackle the process plan search problem used the same problem as in the A* and branch and bound work described in the previous chapter. At that stage of the research a restricted machine shop model was in use. In that model all roughing and finishing operations were compatible, because of this the planning networks and their representation of manufacturing dependencies were not yet in use. Later work on fully incorporating the dependencies into a GA search is presented towards the end of this chapter. However, many important results, which were carried over into the later work, came out of the early investigation; that part of the research is described next.

6.5.2 Early Investigation

Representation

If genetic algorithms are to be applied to a combinatorial problem, the first step is to devise a suitable string representation for the domain to be searched. A very straightforward representation suggests itself for the simplified test problem:

\[ m_1s_1m_2s_2m_3s_3 \ldots m_k-1s_k-1m_ks_k \]

Where, \( m_i \) = machine used for \( i \)th feature
\( s_i \) = setup used for \( i \)th feature

The main restriction imposed by this representation of a plan is the feature order. If the genetic operators are to do their work this order must remain fixed throughout the whole search. This amounts to imposing a strict gene position/plan attribute
mapping. If this mapping were not present, crossover operations would soon produce chaotic and illegal plans. Hence a feature order, which satisfies all the anteriority constraints, is chosen and all strings must stick to it from then on. In the example above the features are actually coded to this order so that the first feature in the order becomes feature 1, the second feature 2 and so on. However, with a subtle objective function, such as \textit{COST}_1 described in an Chapter 4, which transforms the ordering of the features from the initial plan representation to the final plan for execution, this does not mean that the ordering aspect of the problem has been ignored.

A simple ordering strategy was applied which divided into four phases. The features were arranged, using the coding mentioned above, by simply following the order in which they were picked off by the strategy. The four phases are as follows:

1. All those features constrained to to be processed \textit{before} some other feature(s) but not constrained to be processed \textit{after} any other feature(s) are found. These are features associated with the root nodes of the anteriority graph.

2. All unconstrained features are found.

3. All features constrained to be processed \textit{before} some set of features, none of which are themselves ordered yet, but at the same time are not constrained to be processed \textit{after} any as yet unordered features, are found. This involves looping through a list of as yet unordered features until none remain satisfying this condition.

4. All features remaining unordered are collected. These are features associated with nodes at the end of branches on the anteriority graph.

Throughout this procedure checks are made for conflicts in the ordering constraints. As long as the above method is followed, there is an inconsistency in the constraints if a feature chosen to be ordered is constrained to be processed before some other feature which has already been ordered. Under these circumstances the user is told
there is an inconsistency (a loop) in the ordering constraints and this loop is found and displayed.

As already indicated, the costing of a plan represented in this way poses no problems at all - the cost function of Chapter 4, \( COST_1 \), with all its benefits, described earlier, can be used unmodified.

**Operators**

The genetic operators had to be modified to take into account the fact that string members come in tightly bonded pairs. The \((\text{machine, setup})\) combinations represented by adjacent genes cannot be arbitrarily split up or meaningless plan structures will result. The crossover operator must choose crossover points that lie on machine genes, so that its operation will preserve \(< \text{machine, setup} >\) combinations. Two mutation operators were used. If the gene to be mutated is a setup gene then it can only mutate to another legal value associated with the adjacent machine gene. If it is a machine gene then when its value changes to another legal machine for the associated feature, the adjacent setup gene must change accordingly to give an allowed combination. The inversion operator cannot easily be used without elaborate decoding schemes because it destroys the feature order. Hence it was not used in this investigation.

**Results**

The first set of experiments used a randomly generated initial population. A random string is easily generated as follows: for each feature in the previously established feature order, use a randomly generated integer to choose a possible machine from the data structures described in Chapter 4, use a randomly generated integer to pick one of the associated setups. As mentioned earlier, there are several parameters whose values must be adjusted to obtain the best possible performance from a genetic algorithm. The parameters of most interest for this problem were: the
probability of the crossover and mutation operators being used in breeding, PCROSS and PMUT; the population size; and the selection/breeding strategy. Although Grefenstette’s study [41] gives a reasonable indication of the ranges of values to use, the optimal values seem to vary with problem type. A series of experiments indicated the following values were best for PCROSS and PMUT:

\[
\begin{align*}
\text{PCROSS} & = 0.9 \\
\text{PMUT} & = 0.01
\end{align*}
\]

These values were found to be good for a range of different population sizes and were used throughout.

The basic Genetic Algorithm used was the one shown in Figure 6.1. A population of randomly generated strings was produced as described above. The population was stored in an array of five field records. These fields were used for storing the string cost, the string performance value, the assigned number of offspring, the string itself and an associated string of machining costs, one for each feature. This last string was created at the same time as the plan string. It was used to improve the efficiency of the plan costing function. As \(COST(plan)\) moved through the plan string, left to right, it found the appropriate machining cost value by advancing a pointer through the cost string. This is far more efficient than using the look-up tables every time a plan string is costed. Whenever the crossover operator was applied to a pair of plan strings it was also applied to the associated machining cost strings. In this way the correct associated cost strings were automatically produced for the new plan strings.

The performance value of a string was calculated using Equation 6.9.

\[
\text{perf(string)} = \text{maxcost} - \text{cost(string)}
\]

\(\text{cost(string)}\) is the value given by applying \(COST_i\) (Chapter 4) to the string, \(\text{maxcost}\) is the maximum cost of any string in the current population. By subtracting from \(\text{maxcost}\) low cost plans are assigned high performance values, also good and bad quality plans are more clearly distinguished than by using some other method such as inverting the cost or multiplying it by -1 (the usual OR strategy
when costs are to be minimised). The number of offspring, \( N\text{(string)} \), assigned to a string is given by Equation 6.10, \( PSIZE \) is the population size.

\[
N\text{(string)} = \text{round} \left( \frac{\text{perf}(\text{string})}{\sum_{\text{population}} \text{perf}(s)} \times PSIZE \right)
\]

(6.10)

Once all five fields have been filled the breeding process can commence. Each population string has \( N\text{(string)} \) copies of itself and its associated machining cost string put into a temporary population, known as the ‘breeding pool’. The technique for applying the crossover and mutation operators probabilistically was very simple. A random number between 0 and 1 was generated, if this number was less than the assigned operator probability (PMUT or PCROSS) it was applied, if it was more the operator was not applied. Exactly how strings are chosen from this pool for breeding and how the resulting offspring are put into the population is the business of the breeding strategy.

The first breeding strategy used is shown in Figure 6.4. This simple strategy replaced the entire population in each generation. It incorporates the so called ‘elitist strategy’ of ensuring a copy of the best string to date always survives into the next generation. Results of searches using this strategy are shown in Figures 6.5 and 6.6. The results shown are each an average of ten runs using a population size of 60.

The method found a solution with a cost of 301 units, much better than anything found by the branch and bound algorithm of the previous chapter, and far superior to random search. It can also be seen that long searches do not necessarily find better solutions. It was discovered that a few shorter runs, of about 1000 generations, was a better tactic. Every fourth or fifth run found either the 301 solution or one of cost 303. Each run used a different random number generator seed. The characteristic shape of the search is evident in all the graphs. Extremely rapid progress is made in the first 100 or so generations, then a steady pattern of a period of no improvement followed by a sudden jump to a lower cost level is established. The shape of the average cost graphs is very revealing. In every case it is almost identical to the corresponding lowest cost graph, showing that the population very rapidly converges to be almost completely dominated by the best string. As nearly every member of
breed

initialise pointer to population

* main breed

entire population replaced?

yes

no

member of population pointed at best to date?

yes

increment population pointer

pick pair at random from breeding pool

apply genetic operators to produce offspring

replace member of population pointed at, increment population pointer

no

exit

Figure 6.4: Simple breeding strategy.
the population is identical, the crossover operator produces no new offspring and
the search stagnates until the mutation operator throws in some new structures.
The population size had little effect on the search, because of the rapid convergence,
but a value of 60 was found to give slightly better performance, especially near the
start. Although the phenomenon of population convergence was a little worrying,
the typical results in Figures 6.5 and 6.6 show that the search does not come to a
halt and that a good low cost plan is always found. The 301 plan was found on many
occasions but no better plan was ever found. No run of more than 3000 generations
was undertaken. A 1000 generation run with a population size of 60 took about 15
minutes on a Sun 3/50 workstation.

Very encouraged by these results, it was decided to investigate the performance of
a genetic algorithm in which something had been done to stop the convergence.
The first attempt to do this used an adaptation of the simple breeding strategy of
Figure 6.4. In the new strategy a check was kept on the state of the convergence of
the population. This was based around the following inequality:

\[ \text{total\_performance} < \text{PSIZE} + \text{WIN} \]  \hspace{1cm} (6.11)

Where,

\[ \text{total\_performance} = \sum_{\text{population}} \text{performance\_value} \] (total\_performance = 0 for
converged population)

\[ \text{PSIZE} = \text{population size} \quad \text{WIN} = 0.15 \times \text{PSIZE} \quad \text{and} \quad \text{performance\_value} \] is
given by Equation 6.9.

When the condition in Equation 6.11 was true, instead of continuing with the breed-
ing process, a small number (say, 0.05 \times \text{PSIZE}) of randomly generated strings were
introduced into the population and the cycle restarted. Results for this method are
shown in Figures 6.7 and 6.8. Again the graphs show averages of ten runs for popu-
lation sizes of 60. The spikes on the average cost graphs indicate where the random
strings have been introduced. It is fairly clear that this strategy had little effect. The
population very quickly reconverged and the overall performance was very similar.
Figure 6.5: GA results on initial problem
Figure 6.6: GA results on initial problem
to that of the first strategy, if anything slightly worse. Shaking up the population when it was close to convergence simply set up an oscillation effect, the population bounced back towards the converged state, was shaken up, bounced back and so on. This seems to have disturbed the good effects of mutation in this situation and resulted in the slightly poorer performance. Experimenting with different values for the new parameters introduced by this strategy had little effect.

Better results were obtained by using a breeding strategy which was a simple, but subtle, adaptation of the first method. This third strategy did not use 100% population replacement. The algorithm was exactly the same as that shown in Figure 6.4 except that the loop stopped when a certain percentage (less than 100) of the population had been replaced. The new strings were always replaced in a simple linear way starting at the head of the population array. This meant that random strings were constantly present at the bottom of the population throughout the search. With a replacement proportion of 75% a sufficiently large number of random strings remained in the population to allow some of them to maintain an involvement in the breeding process. This effectively maintained a background noise level throughout the search and prevented the population from ever coming anywhere near convergence. Instead of leaving it fixed throughout the GA run, this 'buffer' of poor strings was itself subjected to slow random changes provided by occasional mutations and random string replacements. Results for this method are shown in Figures 6.9, 6.10, 6.11, 6.12, and 6.13. The shape of the lowest cost graphs is similar to all the previous ones but the effect of the buffer of random strings is evident in the average cost graphs. The violently spikey oscillations are in marked contrast to the static convergent graphs. Whereas the best to date may not improve for long periods, the search is throwing up new structures all the time due to the participation of the random strings. Figures 6.9 and 6.10 show results for a population size of 20, Figures 6.11 - 6.13 show results for a population size of 60. Clearly the larger population, ensuring a wider range of allele values, gives a better performance. Searches with this method and a population size of 60 were able to find solutions with a cost less than 300. On several runs a string of cost 297 was found, on others strings of cost 296 and 295 were found. Figure 6.13 illustrates the
Adapted Simple Breeding Strategy used, Population Size = 60

Figure 6.7: GA results on initial problem with simple adapted breeding strategy

Minimum Cost v Number of Gens

Average Cost v Number of Gens

(c)

(d)
Adapted Simple Breeding Strategy used, Population Size = 60
fact that very long runs were not guaranteed to produce spectacular results. Again
it was found best to do several shorter runs, in this case about 3000 generations. At
least one run would find a very good string of cost 300 or less.

At this stage the work was considered successful; a robust method had been de-
veloped to find low cost plans and the population convergence problem had been
overcome. After the search was over, a decoding function was used to display various
members of the population in the form shown below, i.e. an ordered set of ordered
lists of features to be processed on particular machine/setup combinations. This is
the 295 string found by the last version of the genetic algorithm. The core of the
decoding procedure is obviously just the $COST_1$ objective function.

Best String Cost = 295
Sets of features to operate on:
1) setup = 1 , machine = 3
   [ 0 6 7 9 ]
2) setup = 7 , machine = 2
   [ 1 ]
3) setup = 14 , machine = 2
   [ 2 12 ]
4) setup = 1 , machine = 4
   [ 3 4 5 8 10 11 14 15 16 17 ]
5) setup = 6 , machine = 7
   [ 13 ]
6) setup = 4 , machine = 7
   [ 18 19 ]

In all of the searches undertaken none of the randomly generated plan strings had
costs less than 360, most were in the region 380-450. These figures tally with the
random search runs mentioned earlier. Here pure genetic search has produced a plan
with a greatly reduced cost compared with random search. It has minimised cost by
grouping features on the same $(machine,setup)$ combinations, hence reducing setup
operations. If the groups are processed in the order given, no anteriority constraints
are broken. A 3000 generation search with a population size of 60 took about 45
Figure 6.9: GA results on initial problem with added noise breeding strategy, psize=20
Figure 6.10: GA results on initial problem with added noise breeding strategy, psize=20
Figure 6.11: GA results on initial problem with added noise breeding strategy, psize=60
Figure 6.12: GA results on initial problem with added noise breeding strategy, psize=60
Figure 6.13: GA results on initial problem with added noise breeding strategy, psize=60
It was decided to investigate the effect of using constructive heuristics to generate members of the initial population.

6.5.3 Constructive Heuristics Generating Initial Population Members

By a constructive heuristic I mean a function, involving relatively little search, which builds up a plan string according to some set of rules. I have already mentioned one such function in the previous chapter. This was the simple function used to set an initial bound for the branch and bound search. It built a plan by taking a given legal feature order and assigning to each feature the locally cheapest \( (machine, setup) \) combination. This took no account of interactions in the cost functions and produced a plan of cost 340. Some refinements, involving a little more search, gave a function which produced a plan of cost 326. By introducing a single string built up by this function into the otherwise randomly generated initial population, results like that shown in Figure 6.14 were obtained.

By introducing just one string that was considerably fitter than a random one, the search time to find a very fit string was divided by three. Other, more powerful, constructive heuristics were developed. One was a variation on global depth first local best first search. This was able to construct plans of costs around 320-330. The most powerful functions were variations on an algorithm developed to minimise the total setup cost. This was achieved by minimising the number of setups performed, which in turn means minimising the number of \( (machine, setup) \) combinations used and avoiding break-constraints. These constructive heuristics were found to be of
Figure 6.14: GA search with single constructive heuristic, population=60.

extremely high performance, they will be described in detail later in the chapter. Some versions could build strings of costs less than 300. The lowest cost string built like this was a 298 string. An initial population of the following constitution was generated: for a population of 20 - one locally cheapest, eleven using various versions of an algorithm which minimised setups, three using various versions of global depth first local best first search and five randomly generated. Very quickly the GA found a better string, of cost 291, than produced by any of the heuristics. No better plan was ever found and I believe this one is optimal.

The heuristic search algorithms are useful in their own right. They find good solutions fast and, when used to seed the initial GA population, guide the GA to very rapid and even better results. However, they are heuristic and therefore brittle; they blindly follow their particular heuristics, which are not always useful. The GA is much more general and robust; the heuristically generated strings will be discarded if they are not useful in a particular problem. As will be seen later in this chapter, the GA performed very well on much bigger examples of the problem, whereas the performance of the heuristic methods, while still good, decreased relative to the GA.
Restrictions

The initial investigation into the use of GAs in the process plan optimisation problem showed the technique to be far more successful than both the A* and branch and bound searches attempted earlier. Many useful results, such as how to deal with convergence and what values to use for the various GA parameters, as well as the development of the powerful constructive heuristic methods, were discovered during the course of the work. However, the test problem used was rather simplistic. Operation dependencies were not taken into account, the machining cost values were not very realistic and the number of alternatives, although large, was restricted. Once the plan space generator had been fully developed, incorporating the planning networks and using a far more realistic machining cost rule base, a second investigation was carried out. This used the same test component and anteriority graph as before. The machine shop model used was larger, and several representative complexities were added to the component in the way of feature interactions, hence the search space generated was extremely large. A suitably adapted GA search, based on the techniques developed in the earlier research, was successfully applied to this second problem. That part of the research is presented next.

Recently the experiments were re-run using a distributed GA. The quality of the best solutions found was the same, but good solutions were found with far fewer evaluations, and the variance on the best solution found over several runs was far lower than with the earlier sequential GAs.

6.6 GA Investigation Using a More Complex Problem

The representation used in the earlier GA research cannot deal with the dependencies incorporated into the planning networks described in Chapter 3. and fully represented in the second test problem. Consider the following crossover using the
If, in fact, \((\text{machine/setup})\) combinations \(m_2s_2\) and \(m_ks_k\) belong to related roughing and finishing methods as do \(m_{21}s_{21}\) and \(m_{k1}s_{k1}\), then the crossover will produce illegal plans if \(m_2s_2\) and \(m_ks_k\) or \(m_{21}s_{21}\) and \(m_{k1}s_{k1}\) represent incompatible roughing and finishing methods.

To overcome this problem a new representation, using bound triplets of genes, was devised:

\[
f_1 m_1s_1 f_2 m_2s_2 G f_3 m_3s_3 f_4 m_4s_4 f_5 m_5s_5 G \ldots
\]

Related manufacturing methods are grouped together, each group is terminated by a special symbol (\(G\) in above example). As long as the group terminators are the only legal crossover points, the crossover operation will always produce legal plans. All strong relationships between manufacturing methods can be dealt with in this way. The first member of each group is the dominant feature. That is any changes within the group will be with reference to this feature. These features will generally come from the root nodes of the planning networks. So all finished features will be group leaders with their associated intermediate features in tow. This representation will be referred to as the \(\text{grouped feature representation}\).

The basic GA used was the same as for the first study. However, the details of
many of the functions became far more complex in order to accommodate the new representation with its explicit dependencies. Indeed, a great deal of effort was put into rewriting the optimisation software. The features were still given a strict order from which the GA worked. This was again done using the method described earlier in this chapter. This underlying feature order had now to be mapped to a grouped order to allow the plan strings to be built up and used. As before, the features were coded from 1 to N, where N was the number of features to be processed, in accordance with the initial order. The machining information for the features was put into the manufacturing_info array (see Chapter 4) following this order. The grouped feature order was generated by moving through the manufacturing_info array looking for dominant features, that is features with dependent (intermediate) features. The groups were built up by following the dominant features with their dependent features, according to the initial order. The strings were costed by mapping them back to the initial order, in other words mapping them to the original linear representation, and then using $COST_1$ as before. To illustrate this, consider the simple case shown in Figure 6.15.

This shows five features and their interdependencies, the nodes contain feature,machine,setup triplets. Features 1 and 2 are dominant and the rest are dependent. The features
have been ordered according to their anteriority graph and then coded to this order. That is, 1 is the feature code for the first feature in the order and 5 the code for the last. The grouped order built from this situation would be: 1,4,6 G 2,3,5. The grouped feature representation and its mapping to the linear representation for costing is shown in Figure 6.16. The associated machining cost string also followed the grouped feature representation and was used in much the same way as before.

All of the string building functions are complicated by the grouped representation. Machines and setups chosen, either randomly or heuristically, for dependent features must be compatible with those already chosen for the relevant dominant feature. As mentioned earlier, this is facilitated by the data structures used to store the search space.

The mutation operator is similarly complicated. Any mutations to machine genes associated with dependent features must be in accordance with the values associated with the dominant feature for that group. If a machine gene associated with a dominant feature is mutated the machine and setup genes for all the dependent features must be mutated to bring them into line.

The combination of this representation and the subtle objective function $COST_1$, means that computationally very inexpensive genetic operators can be employed while still allowing the ordering and machine/setup selection dimensions of the problem to all be searched simultaneously. The alternative would have been to have used
a simple direct linear plan representation and a very straightforward cost function such as $COST_0$ from Chapter 4. The ordering aspect of the problem would have to have been explicitly handled by allowing arbitrary feature orderings on the plan genotypes. This in turn would have led to the use of very computationally expensive genetic operators to ensure two string recombine to produce a legal offspring. This is non trivial, since it must be ensured that features do not appear twice, or not at all, after crossover, and that ordering constraints and dependencies between operations are not violated. Clearly simple crossover between two strings with different feature orderings would nearly always result in an offspring violating one or more of these conditions. Hence, for instance, either expensive repair mechanisms or recombination operators which check for all possible constraint and dependency violations would have to be used. The approach used here avoids all these problems.

**Results**

Results for GA searches using the grouped feature representation and randomly generated initial populations are shown in Figures 6.17 and 6.18. The added noise breeding strategy was again found to be best. The same values for PMUT and PCROSS that were used in the earlier research were again found to give good results. It was found that population sizes larger than 60 gave no discernible increase in performance so this value was retained. The shapes of the graphs of results were very similar to those of the previous experiments.

Using the analysis of Chapter 4, the size of the search space was estimated to be $3.4 \times 10^{20}$ for the first test problem and $2.5 \times 10^{51}$ for the second test problem. Because the second problem is so much larger it is not surprising to find that the GA must be run longer to achieve good results. With the greatly improved machining cost rules used in the second test problem, randomly generated plans were typically costed at 8,000 units, indeed a random search through 10,000 strings never found one costing less than 7,600. Such a random plan is shown in Figure 6.19. The GA search using a randomly generated initial population and runs of about 5,000
generations was always able to find many plans of cost 4,000 units or less. Longer runs, of about 10-15,000 generations, were able to find several plans of cost 3,200 units or less. One of these plans is shown in Figure 6.20. Again it was observed that very long runs were not guaranteed to give the best results. Because of the size of the search space and the relative decrease in cost achieved, the results for the second set of experiments were even more impressive than the first. Again, recent runs with a distributed GA gave as good results in fewer evaluations and with much lower variance on best solution found over several runs.

Heuristically Generated Initial Populations

Again tests were performed with heuristically generated initial populations. The three basic methods used were the same as in the earlier investigation, namely: locally cheapest operations heuristic, minimise setups heuristic and global depth first local best first search.

Locally Cheapest Operations Heuristic
This very simple heuristic is represented in Figure 6.21. It moves through the pre-assigned feature order choosing the cheapest possible machine and associated setup for each feature, irrespective of any global considerations.

Minimise Setups/Maximise Size of Grid Sets Heuristic
This heuristic is more complex, its algorithm is outlined in Figure 6.22. It makes use of the cost function's simulation grid (see Chapter 4) trying to maximise the grid set sizes and hence minimise the number of setups. Several versions of this heuristic were used. A single pass version and various multi-pass variations. The multi-pass methods moved features between the sets trying to minimise the number of sets until an equilibrium was reached. Some of these algorithms started at the beginning of the grouped feature order, some mid way through on the first pass, some chose the first machine/setup combination randomly, some using the locally cheapest operation method. They were able to find better results than the first heuristic.
Added Noise Breeding Strategy, Population Size = 60
Grouped Feature Representation used

Figure 6.17: GA results on harder problem with added noise breeding strategy, psize=60, more complex representation used
Added Noise Breeding Strategy,
Population Size = 120 (graphs (a) & (b)),
Population Size = 60 (graphs (c) & (d))

Figure 6.18: GA results on harder problem with added noise breeding strategy, psize=60, more complex representation used.
Plan String (54) Cost = 7992
Sets of features to operate on:

1) setup = 6, machine = 6 [ 0 ]
2) setup = 5, machine = 4 [ 1 ]
3) setup = 2, machine = 3 [ 2 ]
4) setup = 19, machine = 8 [ 3 ]
5) setup = 17, machine = 6 [ 4 ]
6) setup = 21, machine = 2 [ 5 ]
7) setup = 4, machine = 6 [ 6 ]
8) setup = 3, machine = 10 [ 7 ]
9) setup = 3, machine = 5 [ 8 ]
10) setup = 7, machine = 9 [ 9 ]
11) setup = 4, machine = 9 [ 10 18 ]
12) setup = 1, machine = 2 [ 11 ]
13) setup = 2, machine = 3 [ 12 ]
14) setup = 4, machine = 5 [ 13 ]
15) setup = 3, machine = 4 [ 14 ]
16) setup = 8, machine = 8 [ 15 ]
17) setup = 2, machine = 9 [ 16 ]
18) setup = 7, machine = 11 [ 17 ]
19) setup = 3, machine = 8 [ 19 ]
20) setup = 17, machine = 2 [ 20 ]
21) setup = 21, machine = 3 [ 21 ]
22) setup = 1, machine = 11 [ 22 23 ]

Figure 6.19: Randomly generated plan from initial population
Best String Cost = 3011
Sets of features to operate on:

1) setup = 5 , machine = 6  [ 0 1 3 9 15 ]
2) setup = 1 , machine = 4  [ 2 6 7 8 10 11 12 13 14 16 18 19 ]
3) setup = 17 , machine = 2  [ 4 20 ]
4) setup = 21 , machine = 2  [ 5 21 ]
5) setup = 5 , machine = 11  [ 17 ]
6) setup = 1 , machine = 11  [ 22 23 ]

Figure 6.20: Highly fit plan found by GA
Figure 6.21: Local cheapest heuristic
Minimise Setups

Take next feature to be processed

Find largest available set on the grid

Long_len_so_far > 0?

Pick mc/stp combination for the feature

Find length of lists of features at (mc,st) coordinate on grid

If length > longest length so far, save mc,st, set longest len = length

Add feature to appropriate list on grid

Find cheapest local mc/st comb. add ft to appropriate list on grid

* Figure 6.22: Maximise grid set heuristic

148
Global Depth First Local Best First Search

The algorithm for this method is shown in Figure 6.23. Features are taken one at a time, strictly according to the pre-assigned order, giving the depth first element. The machine/setup combination which gives the cheapest partial plan is chosen, giving the best first element. Partial plans are costed using \( \text{COST}_i \). The algorithm is similar to the branch and bound method described in Chapter 5. It involves by far the most search of the three techniques and was able to find the lowest cost plan. However, note this is not as good as plans found by GA search with a randomly generated initial population.

Results from Heuristically Generated Initial Populations

Figure 6.24 shows a GA search with one string of the initial population generated by the locally cheapest operation heuristic and the other fifty nine generated randomly. The addition of this one string has significantly accelerated the GA's descent into the lower reaches of the cost graph. Using an initial population of the following profile: one locally cheapest string, four from various versions of the minimise setups method, four from global depth first local best first search and fifty one randomly generated, a GA search exchanged information between strings built up by the various methods to very quickly find plans of lower cost than any found by the individual heuristics. Many of these plans were also of lower cost than any previously found using a purely random initial population. These plans were always discovered after ten or less generations. Much longer runs never found any cheaper plan although many costing less than 3,200 units were found.

6.7 Summary

To summarise, the GA search with the grouped feature representation acting in the extremely large search space of the second test problem gave very good results. It was able to find lower cost plans, starting from a population of randomly gener-
Figure 6.23: Global depth-first, local best-first heuristic.
Figure 6.24: Results of run with single cheapest machine heuristic used.
ated legal plans, than any of the constructive heuristic search methods developed, although some of these were able to find very low cost plans themselves. When the GA was applied to an initial population including strings generated by the heuristics, it combined information from the various heuristics to give significantly lower cost plans than by any of the heuristics used by itself.

Similarly good results have been found with many other components.

This chapter has demonstrated the power of GAs when applied to the single process plan optimisation problem. But what of the more general problem of taking into account interactions with other plans?