Model Generation using Genetic Programming

H.Glaser[+], D.De Roure[+], J.Putney[*], and A.Salhi[+]

[+] Department of Computer Science and Electronics,
The University of Southampton,
Southampton SO17 1BJ, UK

[*] Trading and Planning Group, National Power PLC,
Windmill Hill Business Park, Whitehill Way,
Swindon SN5 6PB, UK

Abstract

In search and optimisation applications, model building is largely manual. However, relationships binding variables and making up constraints may be automatically generated if a complete enough body of data is available. At present, at the level of most businesses such a body is not only available but unexploited.

In the following we shall rely on these data which ultimately can be used in the constraints description of the problem. The efficient implementation of this process is also addressed.

1 Introduction

An important step in the process of generating applications in the Search/Optimisation domains is the generation of a requisite model. The task of building such a model is tedious, time consuming and prone to errors. It requires an accurate representation of the different entities involved (variable definition), and the pinning down of relationships between them, usually in an algebraic form. One or more of these relationships form the objective, others, the constraints. The latters define the search space.

Some relationships are obvious and easily derived from the problem definition. Others, however, can be difficult if at all possible to find, for we don't know they are there. One way of looking for them is to consider the accumulated data about the application. There is no guarantee that relationships will be discovered at all, and it is not even clear what relationships or models are looked for. However, 'something' may be found if the data is complete enough. Moereover, it is an automatic process and hence by extension, one hopes to generate whole requisite models in a similar way. Note that these relationships can also be used on their own as regression models and it is in this respect that they are considered here for the time being.

The process is computation extensive which points to the use of novel algorithms and hardware architectures. It is, in the specific form considered here, commonly referred to as Symbolic Regression or Symbolic Function Identification. Note that in the classical regression the model is known beforehand.

2 Context

Recently, businesses have come to realise the potential of the large amounts of data they keep gathering. Although a lot of it is in databases easily accessible through specialised query languages, less organised data is common and until now is only taking space. The trend now is to put all business data in data warehouses aimed at providing decision support. The aim is to be able to extract manageable chunks of information (*Data Summarisation*, see [**?**]) from the organised as well as the nonorganised ever growing heap of data and make sure that it is understandable to humans or precisely to the decision maker.

Decision support systems already achieve this in many ways such as cluster analysis, trend spotting, visualisation of large data, induction and pattern recognition and behavioural modelling. However, we are here concerned with data mining in its narrow and specific sense as matched by the need to extract models used in Search and Optimisation applications.

2.1 Data Mining

It is rare nowadays to open any business oriented publications without finding adverts or articles referring to data mining. It is currently such a fashionable expression that it is attached to a myriad of topics and products. Some manufacturers go as far as to attach it to existing products.

Data mining is certainly attached to databases, data warehousing, image processing, pattern recognition and much more. Because of the various contexts in which it has been used, it is slightly debased. This calls for refocusing the term to what it normally means, and to what it means in our context.

Data mining is three folds:

1. There is the statistical approach, *hypothesis testing*. Here the system is presented with a hypothesis such as 'Those who buy coal have toddlers. Is that so?' More precisely it can be formulated as :'Is there a positive correlation between coal and baby food?' The system is restricted to a certain type of data, which may still be vast, but, nevertheless, is well specified.

2. There is *directed data mining* where the system is steered towards what to look for as in 'Is there a link between spending at a supermarket and residence area or address of customer?' Here the problem is an induction one. The steer however is not as strong as in hypothesis testing. The problem is commonly solved using decision analysis approaches as well as classification newral nets, clustering methods and perhaps regression.

3. There is *pure data mining* where no steering and no constraints are imposed on the system. In this situation, the user does not know what to look for. The problem is in the same spirit as that well contained in the question: 'Is there anything interesting in my data?' It is then hoped that the system will unravel interesting facts about the data that will assist the decision maker.

3 Search and Optimisation: Stochastic Approaches

In search/optimisation problems [**?**, **?**, **?**] the aim is to find the best plan to configure a system. This plan is usually in the form of a vector. Here, however, we are looking for an expression, a model which best represents a given set of data. Ultimately, this expression may be used as a regression model in its own right or as a building block in the construction of constraints of a search/optimisation problem itself.

All expressions, form a search space. To proceed with the search we need a criterion function, or objective function, and a description of the space itself, using limitations or constraints. The objective function provides us with a handle for choice between different members of the search space, and the constraints tell us what is a valid member. Note that constraints may be implicit. For instance, expressions are restricted to those returning values that can be found in the mined data, up to a tolerance.

Even though we have a selection criterion, the objective function, except in special cases where assumptions on it (Convexity, Continuity,...) are made, it may not be possible to guarantee that the global optimum (global optima) is found. Instead, it is practical to settle for the 'best so far' as the candidate solution, usually only a local optimum. The reason is in limitations on the computational expenses allowed, or simply the chosen stopping criteria. Behind it all, still, is the fact that global optimisation peoblems, except in special cases, are notoriously difficult to solve, i.e. it is virtually impossible to construct search algorithms which will guarantee that a global optimum is found. Exhaustive search techniques are especially doomed to failure except when the search space is very small.

Stochastic methods begin with a given candidate, known beforehand or randomly generated in the search space. A move in a random direction is taken from the candidate. If it leads to a better one, then the current point is superseeded by the new, else another random move is operated. Such techniques belong to the hill-climbing, or gradient, family of techniques. Hill-climing, of course, as just sketched will unlikely lead to the global optimum unless the function is unimodal. However, devices which help escape from local optima have been extensively studied as in Simulated Annealing, Multistart, Tabu Search, Clustering algorirthms [**?**].

If the search space is disconnected, some random moves may not be valid. This born in mind, connectivity or connectedness may be assumed and the notion of neighbourhood of a point can be defined. It is technically the set of points falling within a distance δ of a given point. When a point returns the best value for the objective function in its neighbourhood, it is called a local optimum. These points are relatively easy to find as opposed to global optima. Note that a global optimum is also a local one.

3.1 Genetic Algorithms

Genetic algorithms (GA's) are stochastic search methods modelled upon natural selection. They rely on the concept of fitness which measures the success of an individual in reproducing, i.e. passing on its genes to future generations. GA's are an attempt to capture aspects of natural selection essential to their problem-solving capacity. This capacity of GA's has been observed in many areas as will be seen later.

GA's differ from traditional search techniques in that, inherently, they work on discrete spaces. The basic move, unlike hill-climbing, depends on more than one point or individual. In fact a population of individuals is first drawn from the search space, and maintained generation after genaration to be of some size larger than one. Its constituents may change of course.

The move from one generation to another or from one population to the next is effected by the use of appropriate opearators, such as reproduction, crossover and mutation. Note that mutation, as well as reproduction, work from one individual, a single parent, to obtain a new individual. Mutation especially is similar to the move in the hill-climbing approach but is not typical.

In GA's, there is a flurry of selection procedures and operators. Selection pressure is exercised through the careful design of these selection procedures and operators. It is important to note that if good individuals are favoured *a outrance*, early convergence may result into a poor local optimum. This is because the variety in the population which allows the natural selection process to thoroughly cover the search space is fatally reduced by acute selection pressure. On the other hand, no selection at all will result in a very slow convergence which may not be acceptable in practice. It is essential to strike a balance between the number of fit elements in every population and the variety of its pool of genes.

GA's belong to the larger class of Evolutionary Algorithms. Their kin are the so called *Evolutionary Strategy Algorithms* first introduced in Germany in the 60's, [**?**, **?**], where mutation is the typical means of evolution as opposed to crossover in GA's.

Note that the distinction is now so blurred after the modifications brought into boths types of algorithms that it is not ambiguous to refer to them as Evolutionary Algorithms.

4 Genetic Programming: A Review

Genetic Programming (GP) pioneered by J.Koza [**?**, **?**, **?**], [**?**]. is an extension of GA's to operate over spaces whose elements are programmes. It is a randomized, adaptive search method which represents programmes by their parse trees, [**?**]. It allows computers to find solution programmes to some problems without being explicitely programmed to deal with those problems.

A distinguishing element between GA and GP, commonly found in the literature, is that GA's work on constant length *chromosomes* while GP works on a variable length chromosomes. The tree structures processed by GP are of variable length.

A model GP algorithm can be described as folows:

1. Define the set of terminals;
2. Define the set of functions;
3. Construct a fitness function to measure the problem-solving capacity of valid programmes;
4. Choose control parameters;

5. Choose stopping criteria or ways for recognising a solution;
6. Generate a population of programmes;
7. Find the fitness value of each programme in the population. If stopping criteria satisfied go to 10, else continue.
8. Generate a new population by applying some pre-defined breeding operators on the elements of the current population.
9. Repeat from 7.
10. Solution is the programme with best fitness. Stop.

Some of the terms used in this algorithm will be explained later.

4.1 Theory

To those familiar with traditional search and optimisation techniques and their theoretical basis, there is not much theory to speak of in the case of GA's. When GP is concerned, theoretical results are even scarcer. However, there are beginnings in understanding why these algorithms work and where they might not work.

4.1.1 Fitness Landscapes

GA's work on populations of individuals represented by fixed length bitstrings, (chromosomes). If n is the length of a string, the cardinal of the search space is 2^n. If the elements of this space are represented as dots on a plane with distance between every two dots equal to the Hamming distance, i.e. the number of bit changes to transform one string to the other, and if each dot is now raised above the plane by the fitness value of the individual it represents, then hills and valleys will show. This is a *fitness landscape*.

A fitness landscape gives a pointer to how difficult a problem might be to GA's. If it is mainly flat and featureless then the search for a hill top would be very difficult. If it looks like a hedgehog's back, again guaranteeing that a global optimum is found will be almost a lost cause, if on the other hand one or very few hills are apparent then the search may produce a 'good' optimum.

Fitness landscapes are useful when GA's are concerned. They are not so useful however when GP is concerned because there is no equivalent of humming distance when the strings are not bitstrings and are of different lengths to represent the adjacency of individuals. In GP, programmes are represented as trees, genes can be any predefined operand or operator. The adjacency of programmes is rather difficult to catch.

4.1.2 Schemata, the Schema Theorem and BBH

In his ground breaking work [**?**] Holland introduced the notion of a *schema* which is no more than a hyperplane. But the insight was that while explicitely the GA operates on a finite population of individuals, implicitely it processes, in parallel, a very large number of schemata which crisscross the search space. Also, an average estimate of

their fitness is recomputed for each generation. Why are they so important? Because they represent building blocks of fit individuals in the search space, as long as they are short, of low-order and highly fit themselves. What more the Schema Theorem stipulates that the schemata with high fitness are perpetuated more often in the new generations making GA's probably the only class of algorithms which exploit exponential explosion. This theorem is the basis to the now notorious Building Block Hypothesis (BBH) [**?**].

What about GP? The equivalent of a schema in GP is a set of subtrees with common features. This set is infinite but, it is not once a limit is imposed on the size of acceptable trees [**?**], which is the case in practice. There is a difficulty with this notion of schema in GP due to the definition of a schema which is a string built over an extended alphabet with a wild card. This means simply that if the alphabet is, say, 0,1, schemata are built over alphabet 0,1,* where '*' can be '0' or '1'. In the case of GP, the alphabet is made up of higher order entities, such as functions so a schema is unlikely to be a valid programme in the general case. The notion will be viable only under tight restrictions []. Attempts to build a GP BBH on the GA equivalent of the schema theorem have not been satisfactory, although O'Reilly's thesis is a very good attempt, [**?**, **?**]. A GP-hill-climing hybrid algorithm has also been developed and the results point to the superiority of such an algorithm over simple GP of hill-climing simple algorithms.

4.1.3 Minimal Deception

In order to understand what makes a problem difficult to GA's and also due to the shortcomings of Shemata approach and BBH, attempts have been made to construct problems which 'fool' the algorithm so as convergence is never achieved or only at great costs. The work of Goldberg, [**?**, **?**], is the starting point. It is possible to build such problems, but not for all variants of the GA, i.e. given the opportunity, an algorithm may be tuned to deal with problems. The MD idea does not seem to catch on GP.

This lack of satisfactory theoretical tools to make predictions, measure performance and analyse GP as well as GA's is a stumbling block to newcomers to evolutionary approaches. However, efforts are being made to overcome it. Those of Altenberg for instance, [**?**] who considered evolvability as the main ingredient in the success of GP are interesting.

4.2 Practical Aspects of Genetic Programming

4.2.1 Measuring Fitness

The fitness measure is the criterion used to distinguish between programmes. A measure whose sink is of very low cardinality (≤ 5 for instance) will not be suitable as a large proportion of individuals in the population will be of the same fitness; progress towards a fit individual will be very slow. The ideal fitness measure is the one which reveals all differences between any two individuals. If a population is ordered according to it, then every number between 1 and the population size should be allocated. In other words, no clusters of individuals with the same score should

show. Such a fitness measure will be hard to find, but it should be the target when designing such a measure.

A simple measure of fitness in GP is the error measure [**?**]. It simply computes the difference between the output of a given programme and the target output, when it is known. This is usually the case in regression type problems.

Other fitness measures can be the computation time of programmes in the population, or their memory requirements, the number of hits scored given the input to all programmes. This is termed *absolute fitness*. When it is the score of an individual in a subset of the population, which is itself evolving, then *relative fitness* is the term. When programme runtime gives the fitness, some programmes with infinite loops may cause the braek down of the whole entreprise. Measure, such as maximum number of loops, or maximum CPU time allowed may be introduced, [**?**].

A mixture of measures may also be used. *Pareto scoring* for instance uses multiple criteria such as functionality and efficiency to compare programmes [**?**].

4.2.2 Genetic Operators

Reproduction:
Reproduction is an asexual genetic operation. It promotes fitness in the population by selecting individuals using a fitness measure and copying them into the next generation unaltered[**?**, **?**]. The positive aspect of the reproduction operator is that it is not expensive: no fitness is recomputed for the individuals copied into the new population as their fitness is alraedy known. However, it is only best if applied to a low percentage of the population, say 10%.

Crossover:
Crossover is a sexual operation in that it operates on two individuals chosen by some means based on fitness. These two individuals exchange parts, such as branches in the case of tree structures. The result is two children which will join the new population. Unlike reproduction, here the fitness of the offspring has to be computed. This is bad news since the computation of fitness absorbs most of the CPU time for any non-trivial problem.

Self-crossover:
It operates on a single programme as both parents. The selection of the individual can be done through any selection approach.

Cassette-crossover:
This allows subexpressions in the middle of two parent programmes to be swapped. This operator relaxes the standard crossover which permits only subtrees to be swapped. It is however difficult to implement so that the offspring are valid programmes. This operator appears to make gains in the standard GP, i.e. without ADF's [**?**].

Hoist:
It is an asexual operation which randomly choses a subtree from an existing pro-

gramme structure to be included in the new generation. This operator promotes parsimony, [**?**].

Mutation:
Mutation operates on single individuals and thus is asexual. It alters randomly chosen nodes in the tree structure representing a programme. Terminal nodes can be changed into any other terminals but function nodes can only be changed into functions which accept the same arguments.

Mutation promotes diversity in the population: terminals and functions that have been driven out of the pool of the programmes in the current population may be brought back.

A useful variant is *shrink mutation* which takes the subtree with root the node to alter and replaces with it the whole parent. That keeps the size of programmes in check, [**?**].

Mutation is considered a secondary operation compared with reproduction and crossover. Other secondary genetic operators are *permutation, editing, encapsulation and decimation*, [**?**].

4.2.3 Re-usability and Abstraction

Re-usability of modules is important to the success of any high level programming language. The concept is embodied in the notions of *subroutine* and *function*. In GP, it is easy to see how beneficial re-usability can be: simply, sequences of code that are useful need not be rediscovered in different parts of the tree representation [**?**, **?**]. Moreover, recursivity can be harnessed, generalisation of the concepts of hierarchical problem solving may be possible.

Automatically Defined Functions:
In [**?**] ADF's have been extensively studied. They are evolvable modules of an evolving genetic programme. They can be called by the main GP programme itself and used as functional building blocks. To allow these modules to carry on evolving, operators such as crossover are tricky to implement. However, results show that problems not satisfactorily solved with standard GP, have been satisfactorily solved when ADF's were used.

Concepts Reuse:
In [**?**], Seront attempts to introduce libraries of *concepts* which are then used by GP based systems to inject into the population of programmes they are evolving. It is a different view, from ADF's in that it's internal as well as external as subtrees which form a valid 'concept' may be used in a totally different problem. With ADF's subtrees are only used in the course of solving a given problem; ADF's are only for internal use. Moreover an ADF may not be a valid 'concept' in Seront's definition. It is rather akin to system libraries which are familiar.

Adaptive Representation:
Adaptive representation is another attempt to introducing reusability. In [**?**] it has been reported that by analysing the evolutionary trace of GP's, subtrees which in-

crease the fitness of programmes are isolated and added to the set of functions of the main GP programme. Enrichment of the latest generation with newly generated programmes using the extended function set seem to have a positive effect on the overall performance of GP on some prblems.

Other attempts, [**?**], were made in *abstracting data types, ADT's* and, [**?**], in *genetically building libraries, GLiB's*. Results are not conclusive, but it is the direction for fruitful research, [**?**].

4.2.4 Breeding Approaches

Breeding approaches go hand-in-hand with fitness measure. They concern ways by which a new generation is obtained from the current one, i.e. how parents are chosen, which individuals may be replaced and so on. Their ultimate aim, common to GA's and GP, is to ensure a good cover of the search space and avoid premature convergence to poor local optima.

fitness-proportionate selection:
In this approach, selection depends on fitness and chance. A lottery wheel is used, but a bias in favour of individuals proportional to their fitness is enforced [**?**], so that the 'survival of the fittest' concept is implemented. That is to say that if a programme has fitness which is 50then this programme will be represented on the wheel to occupy half of it, 180°. This means it will be picked more often and its genes have more chances of passing to the next generation.

Tournament selection:
Two individuals are picked at random, the one with higher fitness is the candidate to replace an unfit element previously selected or to mate with it. Tournament selection tries to simulate what happens between animals during the mating season. Usually bouts between two individuals are fought to serve a female [**?**].

rank selection:
Here selection pressure is introduced so that among comaparatively fit programmes, those which are dominant are selected. Rank selection exagerates the difference between losely clusterd fitness values so that the better ones are sampled more. [**?**, **?**].

Generational and Steady State Genetic Programming:
Generational GP (GGP) occurs when, population size kept constant, the entire population changes from generation to generation. No elements are spared on any basis. In Steady state GP, however (SSGP), only a small number of elements are replaced from generation to the next.

Demes and Locality:
Demes or islands, are semi-disjoint populations which are allowed to evolve locally, but migratory agents probabilistically selected are allowed to cross the boundaries to mate. This helps avoid early convergence and promotes diversity. Results [**?**, **?**, **?**, **?**] show benefits of the demetic approach. In [**?**] explicite islands of different sizes were used, while in [**?**] the islands are defined using a form of geographic neighbourhood.

Parents are selected from a neighbourhood and the offspring are placed in that area. It is reported that the notion of demes increases the the generality of the whole population and the overall efficiency.

The advange of the demetic approach is its inherent parallelism. In [**?**] linear speed-up has been achieved when this approach is adopted in the implentation of GP on a network of transputers to solve some problems.

In [**?**] it has been suggested that the selection of similar parents from the demes, and crossover points which minimise the difference between the parents and the offspring also lead to beneficial effects.

Elitism:
It is a breeding policy which favours the fittest individuals in the population. Its draw back is that some elements never get replaced as in SSGP. Because of that elitist approaches may cause premature convergence.

There are numerous breeding policies on top of the ones discussed here. Notable ones are *Brood Selection*, [**?**], *Population Enrichment*, [**?**] and *Disassortative Mating*, cite94:Ryan. They all seem to have positive effect on GP implementation results on some problems according to the reports. It remains however to try them on other problems and perhaps carry out a one-to-one comparison to see their individiual merits.

4.3 Applications

Given the wide scope for applying GP, an exhaustive list will be very long. Among the areas in which it has been successfully applied, to name a few, are the following. *optimal control*, *robotic planning*, *playing startegies*, *market strategies* and *evolution of emergent behaviour*. A longer list may be found in [**?**]. Here, our main interest is in *Symbolic Regression*

Symbolic Regression

Symbolic regression, (SR), already attempted in the 50's and 60's, [**?**, **?**], is an important applications of GP [**?**]. It is an attempt to automatically generate models, unknown beforehand, unlike classical regression, to fit data comprising two distinct groups, input data and output data. In more precise terms, the symbolic expression looked for is the one which, given as argument a subset of the accumulated data input, will produce, after evaluation, a result within a tolerance ϵ of a subset of the accumulated data output. In noisy data environments, it is customery to talk of *empirical discovery* and when sequences are involved where the aim is to find a sequence element of a certain index given some previous elements, then *sequence induction* is concerned. They are special cases of SR.

Just as in the model algorithm for GP of Section 4, here SR requires the six key steps for the system to be initiated, namely:

1. Determine the set of terminals;
2. Determine the set of functions;
3. Devise a fitness measure;

4. Determine the stopping criteria;

5. Generate an initial population of programmes.

4.3.1 Set of Terminals

If Θ is the set of terminals, then Θ consists of dependent and independent variables, as well all constants relevent to the problem in hand, (see *sufficiency* of Θ in [**?**]). Terminals form the leaves of the tree structure containing the evolved programmes.

Constants: Constants are particularly difficult to guess. Ways of evolving them have been devised. In [**?**], ephemeral random constants are used to introduce constants into evolving GP expressions. In [**?**] a genetic operator is described which affects only these ephemeral random constants of Koza in [**?**]. The operator is called a *Constant Perturbation Operator*. It serves fine-tuning coefficients in evolving expressions and introduces new terminals in Θ. This is achieved by multiplying all these ephemeral constants by a random number in the range $[0.9, 1.1]$, which is a perturbation of 10

4.3.2 Set of Functions

Let Φ represent the set of functions. Φ consists of the standard operators ($+$, -, $\div$, $\times$) and other fuctions such as sin, cos, exp etc... which may be relevent to the problem in hand, (see *sufficiency* of Φ in [**?**]).

Closure:
This property is required from all evolved programmes. It means that expressions and the different arrangements of them must be valid programmes. All functions in Φ should be handled with attention in all expressions using them. For instance the division operator '$\div$', as it cannot be given '0' as divisor, must be written so that it can handle it if it occurs without causing a crash in the evolved programme.

4.3.3 Fitness measure

As explained in Section 4.2.1., the fitness measure should tell us how good the expression generated is with respect to the set of data we have in hand. Let this expression be $\phi(x)$ and the working data the real values (`float`) stored in arrays `x[]` and `y[]`. The fitness function can be the discrepancy between the values in array `y[]` and the values returned by $\phi()$ when it takes as arguments values from array `x[]`, or the number of values returned within ϵ of those in `y[]`, or some average over differences between values returned and those in array `y[]`. It can be a combination of these ideas.

4.3.4 Control parameters

These parameters control the process of GP. In [**?**], 19 parameters are listed. Obviously, not all of them are required in an application. They depend very much on the breeding policy adopted and the genetic operators. Parameters which are most common and used here are the population size tt PopSz, the proportion of this to be generated, after initialisation, by the genetic operator crossover, `XRate`, the rate

of mutation `MuRate`, the maximum size of any program to be generated , i.e. the maximum size of the tree which will hold it, `MaxTreeSz`.

4.3.5 Stopping criteria

One the most potent stopping criterion is the available CPU time. This is implemented by imposing a limit on the maximum number of populations the GP programme may generate in the course of its run. The number of hits is an obvious stopping criterion as well, because it allows to recognise a very 'fit' expression which may be what we are looking for, it is a *success predicate*. It can be a great saver of CPU time. Other stopping conditions can be devised especially if a lot of information is know about the data mined.

Here, the stopping criterion is either a generated program matches all `y[]` values when we pass to it as argument the values in `x[]`, or the maximum number of generations allowed is exceeded, i.e. `MaxNbGen`.

4.3.6 Initial population

The random generation is the most popular and effective way of constructing an initial population. However, as in the previous section, advantage may be taken of the knowledge of the data to generate individuals likely to fit it. For instance, in market data there is a periodicity element to it which may be exploited by making sure that expressions have trigonometric functions in them.

4.3.7 Basic SR Algorithm

Algorithm

In algorithmic form the steps described above will look like this:

Initialisation:
set `MaxNbGen` the maximum number of generations;
set `PopSz` the size of the population (constant);
generate an initial population of random programs made up of the
elements of the terminal and function sets of the problem;
`NbGen := 0;`

Start:
While (`NbGen < MaxNbGen+1` and No Program is 100 per cent fit) do:
Assign a fitness value to each member of the population
according to how well it solves the problem.
Generate new population from a selected number of programs
according to fitness using the following operators:
1) Copy existing programs to new population;
2)Mutate programs from old population and incorporate
in new population to increase diversity of stock;
3) Create new ones by crossing two randomly chosen programs;
Endwhile

Choose from resulting population the program with best fitness as a candidate solution to problem.
End:

4.4 Implementation

Data Structures: The Tree

The tree structure is the most appropriate data structure for GP. This is due to the fact that unlike the standard GA algorithm, in GP the chromosome has a variable length.

GP-SR is stack based. Programmes are stored in prefix tree structures.

Modules of the GP-SR tool

Beside the C library functions, GP-SR has the following functions:

function `ReadData()` to read control parameter and arrays `x[], y[]`;

function `RandGen()` to generate random trees holding the initial random expressions or programs;

function `fitness()` to find a fitness value of any valid program;

function `Copy()` to move a given program into a new tree structure (may be done by pointer updating);

function `LabelTree()` to associate an index with every node of a given tree so that random choice of a node will be easier to implement;

function `RandomNode()` to find a random node in a given tree;

function `Mutate()` to alter a randomly chosen node to take a new value and keep the program valid (closure);

function `Cancatenate()` to join two subtrees into a valid tree or program;

function `Xover()` to perform crossover between two trees at randomly chosen nodes and ensure that offsprings are valid programs;

function `DispProg()` to output the candidate solution program either as an algebraic expression or as a C program.

4.4.1 Experiments

At the moment, the tool is tried on simple data such as in textbooks, (Koza's examples for symbolic regression). Polynomials fitting the data are evolved by GP-SR. It remains to be tested on larger sets.

4.4.2 Applications of SR

4.4.3 Conclusion and current work

We have considered model generation for search/optimisation in the context of data mining and looked at the most promising approach to tackle it with. This approach, namely genetic programming has been reviewd. The basic components of a tool for symbolic regression have been presented. The tool is still under development and no major experiments have been carried out on real world data yet. In due course the tool

will be tested on data from the electricity market, and because of the computational demands already apparent of the tool, its parallelisation is being investigated.
Cropping: In the process of evolving expressions, extraneous subexpressions are likely to appear. These subtrees will be evaluated for fitness despite them being redundant for instance. They can also propagate in successive populations. If it can be recognised then cropping it prior to fitness evaluation will be beneficial. Subtle approaches are required though in order to recognise a extraneous subtree. It may be just as expensive to find it, or costly if wrongly removed as it may be best to leave the process of evolution to deal with it.
Parallel implementation: It is already apparent that the computational demands of GP-SR can be substantial, as is the case with most genetic processing based software. These demands may be met by both an algorithmic approach (cropping, parsimony, ...) Some of these can be met by the use of parallel/distributed architectures.