

# **A Job Shop Scheduler using a Genetic Tree Algorithm**

## **Literature Review**

David Crafti  
School of Computer Science and Software Engineering  
Monash University  
Clayton, Victoria  
Australia  
[dcra2@student.monash.edu.au](mailto:dcra2@student.monash.edu.au)  
January, 2004

## Table of Contents

1 Evolution.....	3
2 Evolutionary algorithms.....	3
2.1 Genetic Algorithms .....	3
2.2 Genitor .....	5
2.3 Genetic Programming .....	5
2.4 Genetic Tree Algorithms.....	6
2.5 Ensuring Chromosome Validity.....	7
3 Scheduling.....	7
3.1 Open Shop Scheduling.....	7
3.2 Job Shop Scheduling .....	8
4 References.....	9

## **1 Evolution**

Evolution, as proposed by Charles Darwin, is about survival of the fittest. The required elements for evolution to take place are a sample of individuals, an ability to change the makeup of the individuals over a series of generations and an environment that causes only some of the individuals to reproduce. The result of these elements is that over many generations, the individuals will be changed to suit their environment in an optimal way.

## **2 Evolutionary algorithms**

Likewise, Evolutionary Algorithms are in the area of study which uses the principles of Darwin's evolution to search large solution spaces for optimal answers to complex problems. All the fields of evolutionary algorithms work in basically the same way (Koza, 92):

1. Create a sample of X random solutions to a problem.
2. Evaluate the fitness of each solution.
3. Determine which Y of the X solutions should be used to create the next generation
4. Create X new solutions, based on the Y solutions.
5. Until a defined number of generations are done, or a set amount of time has passed or a specified fitness has been reached, go to 2.

One of the extra benefits of evolutionary algorithms over standard search or optimisation techniques is the algorithm's modularity. For each of the stages in an evolutionary algorithm, whether creating the initial sample of candidate solutions, called chromosomes, scoring a generation or creating a new generation, the same (or similar) task is done on every chromosome in the sample. This allows evolutionary algorithms to be multithreaded easily so that they can be run in parallel (Anderson and Ferris, 94).

The ways that this basic algorithm can be implemented are many. The earliest and most basic algorithms based on the above are the Genetic Algorithm (GA) that was pioneered by John Holland from the 1960s (Holland, 75) and Genetic Programming in the early 1970s (Koza, 92).

### **2.1 Genetic Algorithms**

GAs (Holland, 1975; Goldberg, 1989) work by modelling parameters of a problem as bit strings. These parameters can represent integers, floating point numbers, options to be chosen or anything else that is applicable to a problem. The bit strings must each be of a fixed length. The parameters that make up the bit string must have the same length as the same parameter in other chromosomes, but the lengths of the parameters

within one chromosome may vary. The way the GA works then, is that first of all, a random sample of individuals is generated. The individual bit strings, which are called chromosomes, have two main features. The first is their genotype, which is the actual sequence which defines the chromosome. It is called this because of the analogy with a genetic sequence in biology. The second feature is the phenotype, which is the decoded version of the genotype that determines the traits of the individual. With each of the chromosomes in the sample, the parameters are decoded and evaluated by the fitness function to determine the quality of the phenotype. When the fitness has been determined for the entire sample, the next generation is created. This is done by the three operations of crossover, reproduction and mutation. Crossover is performed by taking two fit genotypes, choosing a place along the bit string, cutting each of them at that place and then connecting one string's left to the other string right and vice versa. This produces two new chromosomes, which are a combination of the two parents. Reproduction is simply a matter of passing chromosomes which are judged to be above a certain fitness level through to the next generation. Mutation is then done, though sparingly (typically 1%), and it is done by choosing bits in the bit strings randomly and swapping them.

One of the problems with GAs is that over time, the solution quality can stagnate. It is possible, through constant intermingling of all of the chromosomes to produce a set that varies by very little and where only the low rate of mutation keeps the sample from reaching entropy. The Tabu-based Genetic Algorithm (TGA) (Li *et al.*, 02) helps to counteract this effect by assigning a clan to each chromosome. Also attached is a list of clans with which performing a crossover is not allowed, or tabooed. This ensures that there will remain diversity within the gene-pool, however if some clans are kept entirely sealed off from each other, then there will eventually be a few separate pools of stagnant chromosomes. This is where aspiration come in. Aspiration is a force within TGAs that works to overcome the tabu between clans in cases where offspring between two parents would be particularly good. In these cases, the reproduction is still allowed. The two mechanisms of tabu and aspiration help to maintain diversity and contribute to convergence towards a solution, respectively. In (Li *et al.*, 02) it was found that both the GA and TGA converged towards an optimal solution for the problem of oil tank inspection scheduling, but the TGA reached an answer faster, due to stopping the sample from stagnating too quickly.

Another issue that has been demonstrated with GAs (Anderson and Ferris, 94) is the sensitivity to the rates of each operator. GAs typically employ mutation at a rate of 1% or less, because as with the animal kingdom, a high mutation rate will lead to a degradation in the quality of results. It can cause the algorithm to not converge on a result because individual chromosomes are thrown out into different areas of the solution space too regularly.

## 2.2 Genitor

Genitor (Whitley, 01) is an evolution strategy similar to GAs, except reproduction only produces one child, and that child is placed back into the sample, replacing the worst performing genotype. This means that there are no explicit generations.

## 2.3 Genetic Programming

Genetic Programming (GP) (Koza, 90; Koza, 92) was developed by John Koza in the early 1970s and is the most dissimilar of the evolutionary algorithms in common use today. The purpose of a GP is not to determine the parameters that bring about an optimal result. A GP is designed to determine a *program* that will calculate the best answer to a problem. The candidate programs are stored as trees where operators or functions are non-terminals and parameters of the program are the terminals of the tree. What must first be done in GP is to define the possible range of functions and the number of variables that will be at the terminals. What is then done is a random sample is created and they are each tested against fitness cases. A fitness case usually consists of a series of numbers, one for each variable and an output value. For example, for a candidate program:  $3x^2 + 2y - 6$ , there might be fitness cases of:

x	y	total	
3	4	52	and
2	2	6	

The candidate shown is clearly far off being correct (a correct answer is  $-x^3 + y^3 + xy + x$ ), but depending on

## 2.4 Genetic Tree Algorithms

Genetic Tree Algorithms (GTAs) (Kirley, Newth and Green, 02; Kirley, 02) are a fusion of both GAs and GP. A chromosome in a GTA is a tree composed of nodes that correspond to fixed values. The values can be defined in a lookup table, or stored in the tree, but it is important to understand that no values in a GTA are changed by the genetic operators, and unlike GP trees, the same number of nodes are maintained in every tree. The genetic operators that can be applied to a tree are slightly different to either GAs or GP. The first operator is node mutation. All this does is swap the node values (corresponding to an element in the lookup table) of two random nodes in a tree. The next operator is subtree mutation. This takes two subtrees from the one tree and swaps them. The roots of the subtrees must not be directly related or the operation will be invalid due to one subtree having to be attached to one of its own descendants. The third common genetic operator is the grafting crossover operator. It is similar to the reproduction crossover operator in GAs and GP in that it requires two trees as sources. The difference, however, is that due to the need to maintain a fixed size of the tree, as GAs require a fixed bit string length, and because the fixed sized tree must have one and only one of each node, there is no way to guarantee that a standard crossover would produce any valid children. Because of this, the grafting operator works by taking a subtree from a 'donor' tree and grafting that branch onto a free space on the 'recipient' tree. This will clearly produce an invalid tree, so before the grafting can take place, the nodes that appear in the donor subtree must be deleted from the recipient tree and the validity of the tree ensured. Another feature of GTAs is that due to their tree based homogeneous structure, they can be traversed in multiple ways, meaning that for each genotype, there are typically 3 phenotypes. This means that each individual is the equivalent of three different solutions to the given problem. One of these solutions will tend to be quite better than the other two as only one traversal, or phenotype will have been chosen for fitness, whereas the other two are along for the ride, as the case may be, but can still contribute based on the ability to act as a global search.

Genetic Trees are not inherently modular. A subtree is effectively like a function in conventional programming and the rest of the tree, before the subtree, is effectively the subtree's context, or input. What this means is that while a subtree might specify a given set of tasks in a set order, or a particular mathematical equation, or a division of space, the tasks which the subtree specifies can only be placed relative to what has already been placed, so the outcome might be very different to what would be expected in a modular problem.

What this lack of modularity in relation to the given problem means, is that ANY modification to the genotype of a tree could represent a major change to the phenotype of that tree. Some deviations will tend to make

smaller changes to the phenotype, such as node mutation, or in some cases, subtree mutation (at small subtree sizes), whereas other deviations, such as grafting will generally produce large changes to the phenotype. The size of the changes that are made to the phenotype corresponds, generally, to the size of changes made to the genotype, both in shape and in composition.

## **2.5 Ensuring Chromosome Validity**

One way of ensuring only valid solutions are reached is through the use of a repair function (Michalewicz *et al.*, 99, Bäck *et al.*, 95). A repair function takes a chromosome that is somehow invalid, and makes as small a change as possible to make the chromosome valid. In GP, where every subtree within a chromosome must be of the same type, so that the tree is homogeneous, there is little need for a repair function, unless some numbers are mutated out of specified bounds or some problem specific structure is broken. In GTAs, where the numerical values that are fed into the problem do not exist within the chromosome, a repair function is only needed when the ordering of nodes within a tree break some constraint of the problem. If this happens, it can be a difficult problem to fix, as ordering within a tree must be valid in all three traversals.

Another method that is used is a penalty function for infeasible solutions (Bäck *et al.*, 95). This method allows invalid chromosomes to survive, but a penalty is applied in the fitness function depending upon how far from being feasible the chromosome is. This means that if a chromosome is infeasible, yet is considered fit enough to be reproduced then there must be some particularly fit part of the chromosome that should be preserved if possible. One constraint upon this method is that the best invalid solution must still score worse than the worst valid solution. This is to stop the sample of chromosomes drifting towards being good infeasible solutions.

With GTAs, because no data other than node values are actually included in a chromosome, the best strategy is to ensure that the scheme for decoding the trees allows every tree to be interpreted as valid. This is because rearranging the structure of a tree to find valid answers would be a search problem of its own, as well as lending some unwanted weighting to the shape of the trees. The same can be said if the phenotype of the tree, the one-dimensional string representing the decoded tree were to undergo this process.

## **3 Scheduling**

### **3.1 Open Shop Scheduling**

Open Shop Scheduling (OSS) (Martin and Shmoys, 96; ) is a factorial problem that is well known to be NP-Hard and therefore effectively impossible to find optimal solutions for anything but the smallest sizes. The

problem consists of a shop, or factory that is producing multiple items, or jobs. There are multiple machines in the shop and each job must visit each machine exactly once. Each machine can only work on a single job at a time and each job can only be worked on by a single machine at a time. In the most common variety of this problem, there is no setup time for each task, meaning that it takes no time to prepare a machine for a new job, no time to prepare a job for a new machine, and no time to transport jobs between machines. The problem itself is in trying to have every job finished in as little time as possible. This is known as minimising the makespan.

OSS is a well-studied problem (Jain and Meeran, 98; Fang *et al.*, 93; Gu'eret and Prins, 98) and has been attempted in various different ways. Traditionally, researchers have tried to determine definite solutions to the problem, and for certain cases they have been successful (Jain and Meeran, 98). The problem, though, is that because the problem is NP-Hard, as the problem size is increased to real-world levels, the search space expands factorially, making it effectively impossible to solve. Over recent years, different search strategies have been employed to try to find a good answer to hard problems. These search strategies, such as using GAs and GTAs have worked by using evolutionary algorithms to find as many local optima as possible and to then climb them (in these cases by using genetic operators) to find the best makespan. The problem with these search strategies is that it can be hard to find the global optimum and the better the local optima that are found that are not the global optimum, the less the chance of finding the global optimum in continuing generations.

### **3.2 Job Shop Scheduling**

Job-Shop Scheduling (JSS) (Cheng and Smith, 95; Binato *et al.*, 00) is a problem similar to OSS, but where the order in which a job is processed is taken into account. For JSS, as well as defining the amount of time that each job must spend at each machine, an order must also be specified. This significantly lowers the size of the search space, due to the elimination of many solutions from the OSS search space. It also makes the representation scheme for use in a program potentially harder than an equivalent OSS program due to the need to ensure only valid solutions are evaluated.

## **4 References**

(Anderson and Ferris, 94),

E. Anderson and M. Ferris,

"Genetic Algorithms for Combinatorial Optimization: The Assembly Line Balancing

Problem" in ORSA Journal on Computing , Vol. 6,

The University of Wisconsin, pp. 161-173, 1994

(Bäck *et al.*, 95),

Thomas Bäck and Martin Schütz and Sami Khuri,

"A Comparative Study of a Penalty Function, a Repair Heuristic, and Stochastic

Operators with the Set-Covering Problem,

Math and Computer Science Department, San Jose State University, 1995

(Binato *et al.*, 00),

S. Binato, W.J. Hery, D.M. Loewenstern, and M.G.C. Resende,

A GRASP FOR JOB SHOP SCHEDULING,

AT&T Labs, 2000

(Blum, 02),

Christian Blum,

"An Ant Colony Optimization Algorithm to Tackle Shop Scheduling Problems",

IRIDIA, Université Libre de Bruxelles, Belgium, 2002

(Cheng and Smith, 95),

Cheng-Chung Cheng and Stephen F. Smith,

Applying Constraint Satisfaction Techniques to Job Shop Scheduling,

Carnegie-Mellon University, PA, 1995

(Fand *et al.*, 93),

Hsiao-Lan Fang and Peter Ross and Dave Corne,

"A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Re-Scheduling, and Open-Shop Scheduling Problems" in Proc. of the Fifth Int. Conf. on Genetic Algorithms

Stephanie Forrest ed., Morgan Kaufmann, CA, pp. 375-382, 1993

(Goldberg, 89),

D. E. Goldberg,

"Genetic Algorithms in Search, Optimization and Machine Learning",

Addison - Wesley Publishing Company, Inc., 1989

(Gu'eret and Prins, 98),  
C. Gu'eret and C. Prins,  
"Forbidden intervals for open-shop problems",  
Ecole des Mines de Nantes, 1998

(Holland, 75),  
John H. Holland,  
"Adaptation in Natural and Artificial Systems",  
The University of Michigan Press, 1975

(Jain and Meeran, 98),  
Anant Singh Jain and Sheik Meeran,  
A STATE-OF-THE-ART REVIEW OF JOB-SHOP SCHEDULING  
TECHNIQUES  
Department of Applied Physics, Electronic and Mechanical Engineering  
University of Dundee, Dundee, Scotland, UK, DD1 4HN

(Kirley, 02),  
Michael Kirley,  
"Ecological Algorithms: An Investigation of Adaptation, Diversity and  
Spatial Patterns in Complex Optimization Problems",  
Charles Sturt University, 2002

(Kirley, Newth and Green, 02),  
Michael Kirley, David Newth and David G. Green,  
A Tree Based Genetic Algorithm for Solving Open-Shop Scheduling  
Problems,  
School of Environmental and Information Sciences, Charles Sturt  
University, Albury, NSW, 2002

(Koza, 90),  
John R. Koza,  
"Genetically breeding populations of computer programs to solve problems  
in artificial intelligence" in Proceedings of the Second International  
Conference on Tools for AI, Herndon, Virginia, USA,  
IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 819-827, 1990

(Koza, 92),  
John R. Koza,  
"Genetic Programming: On the Programming of Computers by Means of  
Natural Selection",  
MIT Press, Cambridge, MA, 1992

(Koza, 94),

John R. Koza,  
“Genetic Programming II”,  
The MIT Press, Cambridge, MA, 1994.

(Li *et al.*, 02),  
Sheng-Tun Li and Chuan-Kang Ting and Chungnan Lee and Shu-Ching  
Chen,  
"Maintenance Scheduling of Oil Storage Tanks using Tabu-based Genetic  
Algorithm",  
School of Computer Science, Florida International University, 2002

(Martin and Shmoys, 96),  
Paul Martin and David B. Shmoys,  
“A New Approach to Computing Optimal Schedules for the Job-Shop  
Scheduling Problem” in Proceedings of the 5th International Conference on  
Integer Programming and Combinatorial Optimization, (IPCO) '96  
Cunningham *et al.* eds, pp. 389-403, 1996

(Michalewicz *et al.*, 99),  
Zbigniew Michalewicz and Kalyanmoy Deb and Martin Schmidt and  
Thomas J. Stidsen  
"Towards Understanding Constraint-Handling Methods in Evolutionary  
Algorithms" in Proceedings of the Congress on Evolutionary Computation,  
Volume 1  
Peter J. Angeline and Zbyszek Michalewicz and Marc Schoenauer and Xin  
Yao and Ali Zalzala eds., IEEE Press, Washington D.C, pp. 581-588, 1999

(Newth, 02),  
David Newth,  
“Building Blocks and Modules – Some Mechanisms for Adaptation in  
Complex Systems and Evolutionary Computation”,  
Charles Sturt University, 2002

(Ombuki and Ventresca, 02),  
B. Ombuki and M. Ventresca,  
“Local Search Genetic Algorithms for the Job Shop Scheduling Problem”,  
Department of Computer Science, Brock University, 2002

(Whitley, 01),  
Darrell Whitley,  
"An Overview of Evolutionary Algorithms: Practical Issues and Common  
Pitfalls" in Information and Software Technology, number 14, volume 43  
Colorado State pp. 817-831, 2001