

A Job Shop Scheduler using a Genetic Tree Algorithm

David Crafti
School of Computer Science and Software Engineering
Monash University
Clayton, Victoria
Australia

dcra2@student.monash.edu.au

January 29, 2004

Abstract

Job Shop Scheduling is a well explored, yet extremely difficult problem. Genetic Tree Algorithms are a new and promising evolutionary technique to minimise the complexity of difficult search problems. In this thesis, we devise and implement a scheme to use a Genetic Tree Algorithm to try to find optimal solutions to hard Job Shop Scheduling problems. Mixed results are produced, though potential is found in using a Genetic Tree Algorithm for Job Shop Scheduling in its current and slightly modified forms.

Table Of Contents

1	Introduction.....	3
2	Background.....	3
2.1	Evolution.....	3
2.2	Evolutionary algorithms.....	3
2.2.1	Genetic Algorithms.....	4
2.2.2	Genitor.....	5
2.2.3	Genetic Programming.....	5
2.2.4	Genetic Tree Algorithms.....	6
2.2.5	Ensuring Chromosome Validity.....	7
2.3	Scheduling.....	8
2.3.1	Open Shop Scheduling.....	8
2.3.2	Job Shop Scheduling.....	8
3	Representation Schemes.....	10
3.1	OSS.....	10
3.1.1	Weighting.....	10
3.1.2	Genetic Operators.....	11
3.2	JSS.....	12
3.2.1	Scheme 1: OSS Minus Invalid JSS Solutions (OMIJS).....	12
3.2.2	Scheme 2: JSS with Multipass Solution Construction (JMSC).....	13
3.2.3	Scheme 3: JSS with Implicit Task Numbering (JITN).....	14
4	Experimental Setup.....	14
4.1	Genetic Operators.....	14
4.1.1	Calibration Test 1.....	15
4.1.2	Calibration Test 2.....	16
4.2	Weighting.....	18
4.3	Experimental Procedure.....	19
5	Experimental Results and Discussion.....	20
5.1	Experiment 1 - abz6 dataset.....	20
5.2	Experiment 2 – la24 dataset.....	21
5.3	Experiment 3 – la25 dataset.....	22
5.4	Experiment 4 – ft06 dataset.....	23
6	Conclusion.....	23
7	References.....	25
	Appendix A – Main Program Code.....	28

1 Introduction

The project that is to be undertaken for this thesis is to design and create an algorithm to implement a Job Shop Scheduling (JSS) program using a Genetic Tree Algorithm (GTA). Numerous JSS programs algorithms have been implemented, using many different strategies (Cheng and Smith, 95; Jain and Meeran, 98; Binato *et al.*, 00), but due to GTA's newness, the closest application for which a GTA has been used is an Open Shop Scheduling (OSS) program (Kirley, Newth and Green, 02). The results from that paper were very promising, so this project has been undertaken to evaluate the quality of results that can be obtained using a GTA for a JSS program.

This thesis initially reviews many of the relevant areas such as evolutionary algorithms and scheduling problems in Chapter 2 and progresses, as an intermediate step, to describing an OSS program that was created, briefly, as a stepping stone to a JSS program. This will be done merely to put the foundation in place for the JSS program which is seen to be a constrained version of an OSS version.

In order to design an algorithm for either OSS or JSS, representation schemes are first developed and examined in order to foresee any pitfalls that could possibly arise. The main issues that are examined relate to the structure of the tree, the genotype, and how that structure will be decoded to represent a valid solution, the phenotype. Other issues relate to how the trees will be scored, and what weightings will be applied to the scores to determine how many times each chromosome will be used in creating a descendant for the next generation. These issues will be looked at in Chapter 3, followed by experimental setup in Chapter 4, including calibration of genetic operators and weightings coming in sections 4.1 and 4.2, respectively. After experimental setup, there will be experimental results and discussion in Chapter 5, followed by concluding remarks in Chapter 6.

2 Background

2.1 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.

2.2 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).

2.2.1 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.

2.2.2 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.

2.2.3 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 the average fitness, might still be used in breeding the next generation. The way crossover operations are done in GP is to select one branch from each of two parent trees and swap them, producing two new programs for the next generation. This is the equivalent, depending on the traversal scheme used, of doing a crossover operation in GAs where the bits to be crossed over could come from many non-contiguous places. Mutation can also be done in GP, either affecting the type of function to be performed in a non terminal, or affecting the value of a terminal. All subtrees within a GP tree are well formed trees of the same type as the overall tree. This is a must in GP as it allows reproductions to be done without concern over invalid types. A well known problem with GPs is bloating (Koza, 92; Koza, 94), due to the tree structure of the program. This can happen when, over generations, a tree picks up bigger branches in reproductions than it gives. It can allow a single tree to grow very large, with mostly redundant nodes. The nodes will be redundant in these cases because if it was doing something significant, then it would probably rule the individual out of contention for breeding due to poor overall fitness.

2.2.4 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.

2.2.5 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.

2.3 Scheduling

2.3.1 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.

2.3.2 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.

3 Representation Schemes

3.1 OSS

3.1.1 Weighting

The representation scheme that was used for the OSS stage of the project was:

Assign each task to a node of each tree. Then, to score a tree, it would first be traversed and for each node that was encountered, the corresponding task would be processed. In order to process a tree, timelines for each machine and each job were kept so that all time could be kept track of. To process a node, the earliest time that the task could be started was determined. The time to start processing a task was determined to be the first gap on the task's job's timeline that was passed the end of the task's machine's timeline.

When all nodes in a tree were placed on the appropriate timelines, the score for that traversal was determined to be the makespan, or the latest time that a task was finished. The score for the tree itself was then set as the lowest score from each of pre-order, in-order and post-order traversal of the tree.

In order to then decide how many times each tree would be able to reproduce into the next generation, a weighting had to be devised. An initial weighting choice that was considered was the current tree's best makespan divided by the sum of all trees' best makespans. It was decided though, that this weighting would be too flat, considering makespans could be in the order of thousands, so that very little extra weight would be given to good results. Normalising the scores to 0 or 1 before doing the above process was then considered, as this would reduce the flatness of the weighting. The part of the weighting that where the score for a tree is divided by the sum of all trees' scores had to be adjusted, as this would produce a higher weighting for trees with the lowest fitness. The adjustment that took place was changing the numerator and denominator of the division from the raw scores to fractions to the power of the score. Using a fraction less than 1 for the base of the exponential ensures that higher scores receive lower weightings than lower scores. The weighting that was used is described in Equation 1, below.

The reasoning behind this weighting is that:

a) The base of the exponential terms is always less than 1 so that as the score for each tree goes up, the weighting for that tree goes down; and

b) Depending upon the value of a that is chosen, the aggression of the weighting towards the lowest scoring, better trees can be adjusted, with a lower value of a corresponding to more reproductions of the fitter trees and a higher value corresponding to a flatter reproduction profile.

$$weights_i = \frac{(0.8 + 0.02a)^{Scores_i}}{\sum_{i=1}^{i=SampleSize} (0.8 + 0.02a)^{Scores_i}} \dots\dots 0 \leq a < 10 \dots\dots\dots \text{(Equation 1)}$$

The problem with this weighting is that even though the a value allows the aggression to be adjusted, there would be problems with floating point precision and underflow, so the following equation is performed in order to overcome this problem.

$$Scores_i = b + RawScores_i - BestScore \dots\dots\dots(Equation 2)$$

This equation starts with all the scores being raw, untreated scores and then subtracts the best score in the sample from them. This allows the weighting in Equation 1 to take less processing time and have better precision with lower bit floating point numbers. The b variable in Equation 2 also allows another way to adjust the weighting that is done in Equation 1 so that the higher the value of b , the flatter the weighting values.

To then assign the number of reproductions that each tree will be allowed, the sample size is multiplied by the weighting for each tree. This can lead to a number that differs from the sample size though, because of rounding issues with fractions being assigned to each tree. The mechanism that was used to overcome this was to assign any extra reproductions to either the best tree or a tree that is not the worst. Any extra reproductions that were assigned would be taken away first from the worst tree, and then from any tree that was not the best.

3.1.2 Genetic Operators

The genetic operators that were chosen for completing the OSS program were the same as in (Kirley, Newth and Green, 02) as described in 2.2.4. Node mutations were implemented in the same way, as was sub-tree mutation, however, the grafting crossover was implemented with a slight difference. Instead of only being able to delete nodes which were either leaves or parents of leaves, the definition of a leaf that could be deleted was expanded to allow any node that had at least one child that had at most one child. The restriction on which nodes could be deleted was created to ensure that no nodes in the tree could end up with more than two children, but this extra case that was added in this implementation still holds to that rule. The one flaw that is left with the deletion of nodes is due to some deletions being valid in the context of which other nodes are also being deleted. This was seen as a minor problem as the program was able to work well without making checks for this case. It is an area which in future could be explored to evaluate whether it would improve the performance of the algorithm. Another reason for not allowing this sort of deletion, and possibly even the case that was made possible in this implementation, by the effect of deleting large blocks of nodes on speciation. Speciation (Newth, 02) is effectively like a description of the phenotype of a tree based on parts of the genotype. The parts of the genotype that tend to have a higher impact on the structure of the tree and therefore the trees traits or phenotype will tend to be further towards the top of the tree, at the places where the nodes are first read. This, of course, does depend on the traversal, as in-order and post-order will tend to read from either side first more that pre-order.

3.2 JSS

In order to implement a GTA to solve the JSS problem, some strategies had to be developed to represent the solutions to the problem. There are many ways to use the fixed tree of a GTA to represent a valid solution to the JSS, and they all rely on various algorithms to maintain their validity. In this section, there will be three representation schemes that were considered and evaluated for implementing a JSS program. Only one of the schemes was implemented, though an improved scheme will also be detailed.

3.2.1 Scheme 1: OSS Minus Invalid JSS Solutions (OMIJS)

The first scheme that was considered, was to use the OSS scheme as far as possible. This seemed like the simplest way of creating a JSS algorithm. In reality it required many ad-hoc changes to the basic OSS algorithm and in the end was infeasible to implement.

The first problem that was encountered was that to process the nodes of a tree in the same way as the OSS solution would require making many trees invalid due to tasks being encountered in the wrong order.

The second problem that was encountered was that for a tree that was valid in pre-order traversal, it was not necessarily valid in post-order traversal. In fact, it would not be possible to devise a scheme that would allow all three traversals to be used.

This was partially solved by defining some invariants of the trees.

For pre-order and in-order traversal, the following recursive invariants hold:

- 1) The left side of a tree is processed before the right side.
- 2) The right side of a tree is processed after the left side and root node.

Clearly though, for this situation, post-order traversal would have to be disregarded as the invariant that the right side is processed before the root node would conflict with invariant 2, above. This is not a major problem, except that it does interfere with the GTA's ability to do global searches by cutting out one of the phenotypes for each genotype.

Once this partial solution was devised for this scheme, it became obvious that there would need to be more work done in order to allow the genetic operators to work properly. If the operators could not maintain valid trees, a repair function would need to be devised to fix the trees after operations took place.

The first problem with the genetic operators is for the simple node mutation function. It would have to be ensured that the mutation did not break the ordering of the nodes within a job. This seemed unlikely but possible to be fixed.

The next problem was with the grafting crossover operation. In order to graft a branch from a donor tree to a recipient tree, the nodes that appear in the donor branch have to first be removed from the recipient tree. This can introduce a problem due to the ordering in JSS. The problem is illustrated in

Figure 1. If there is an ordering such that 0 comes somewhere before 2 comes somewhere before 5 and that in the tree this is stored as the first tree and then 2 is deleted, then there is nowhere to add the 2 to the second tree.



Figure 1

This problem could be overcome by trying to graft on the new branch and if it could not be done, then to just clone the tree into the next generation. The problem might also be reduced in importance considering the donor branch itself would most likely have almost the same nodes as the nodes which would be deleted due to the fact that the donor branch must have come from a valid tree and that there would be fewer valid trees due to the ordering constraints.

Yet another problem with this scheme was that even though the nodes might be processed in an order set by the tree, this order was not analogous to the chronological order that the tasks would appear on the timelines.

Unfortunately, this ad-hoc scheme for simply updating the OSS algorithm is fundamentally flawed and for this reason, the two other schemes were the only schemes to be considered worthy of implementation.

3.2.2 Scheme 2: JSS with Multipass Solution Construction (JMISC)

This scheme is similar to OMIJS, but with two small differences, which makes the whole scheme act differently. Instead of trying to construct repairing functions or ad-hoc code to repair the obvious flaws with OMIJS, a multipass system was used. First of all, all three traversals can be used for this scheme. In order to score the fitness of any given tree, for each traversal, the tree is converted into a one dimensional array of the task numbers. Then, for each task, it is either processed if all the previous tasks in the job have been processed, or it is copied into a buffer to be copied back once all of the tasks have been processed or copied. Then repeat the process until there are no more tasks. This is the multipass system, which is the first difference. The second difference is that instead of having the machine timelines always increasing, and the job timelines able to have gaps that can be filled by a later task, the jobs timeline must always be increasing. This ensures that as the tasks are processed in the same chronological order as specified by the order defined in the input to the JSS program. The genetic operators for JMISC are the same as those used in the OSS stage of the project.

This scheme effectively means that:

- 1) Any tree is valid.
- 2) There can be multiple genotypes that refer to the same phenotype.
- 3) A tree is not necessarily processed fully in the order it is traversed.
- 4) Small changes in the tree are able to cause large changes in the phenotype, but it is less likely as a node that is added halfway into the genotype might be ignored until the tree is half processed. This allows for an efficient local search.

JMSC is the scheme that was implemented and is evaluated in this thesis.

3.2.3 Scheme 3: JSS with Implicit Task Numbering (JITN)

This scheme is quite different to the two schemes defined above. Rather than have a node in each tree that corresponds to each task in the problem, where there are m machines and j jobs, JITN has m nodes of each of the j jobs. Each of these nodes still has a unique identifier so that for the grafting crossover operation, it is known which nodes should be deleted from the recipient. The big change in how this scheme works is that the tree can be processed in the order in which it is traversed. This is because when a node from job x is reached, there is really no need to know which task it represents explicitly. It is enough to know that it refers to the task whose turn it is to be processed for the given job. This removes the need to have a multipass system as in JMSC.

4 Experimental Setup

All tests and experiments for this thesis were run on an AMD Duron 750MHz processor with 256MB RAM. The programming language that was used for the project is Borland Delphi 5, so that the program could have a graphical front end. All of the tree structures that are used internally use the MS XML Document Object Model. This, combined with the graphical front-end of the program, cause the program to run very slowly, but the quality of results is not effected.

4.1 Genetic Operators

The effectiveness of an evolutionary algorithm is strongly affected by the proportions of genetic operator usage. In traditional GAs, mutation rates are often as low as 1% and can go as high as 15% (Ombuki and Ventresca, 02). Any higher than this would begin to produce random-looking results, as would a severe radiation storm on Earth. This, however is when the mutation is used to change a value in a bit string, independent of the representation scheme that is used. With GTAs, the equivalent operator, node mutation is the swapping of 2 fixed numbers, so the figures from genetic algorithms are unable to be of assistance for a GTA. Also, the mutation rates in GAs refer to how many of every 100 bits are mutated, as opposed to the GTA operator which is only one operation per tree that is chosen for mutation. No further operations are done on a mutated tree. The other two operators, sub-tree mutation and grafting crossover are unlike those in other evolutionary algorithms. Due to the lack of

knowledge about what are good genetic operator rates for a GTA, some exhaustive sensitivity analysis was done to determine the best calibration for the system.

4.1.1 Calibration Test 1

Test 1 was carried out on the standard job shop scheduling test set ft06. The test was set up with a sample size of 50, and values of a and b of 5 and 1, respectively. The rate for node mutation varied between 0 and 0.4 in 0.05 increments. The rate of sub-tree mutation varied between 0 and 1 in 0.1 increments and the rate of grafting varied between 0 and 1 in increments of 0.05. The 3 genetic operators were constrained so that the rates of all 3 summed to 1. The test was then run using these rates for 200 generations each. The numbers that were then plotted are figures that are subtracted from 250. The reason for the subtraction is to allow the results to be seen more clearly on a 3 dimensional graph, as peaks are easier to make out than troughs, so while the best result is the one that took the least number of generations, it will appear as the highest peak on the graph. According to (Ombuki and Ventresca, 02), the optimal value for ft06 is known to be 55, however after running this test briefly, it became apparent that the optimal value was at most 54 as this result was achieved quite often with many combinations of genetic operator rates. A score of 50 corresponds to a test run taking 200 generations to reach the new optimal value of 54, as a score of 200 corresponds to a test run taking just 50 generations. A score of 0 corresponds to the test run never reaching the optimal value. Two test runs were done and the results averaged in order to achieve results which are more statistically significant. More test runs would have been done, except time did not permit, due to how long each test run took and due to the testing computer being 4 years old.

The results of the test run are as follows:

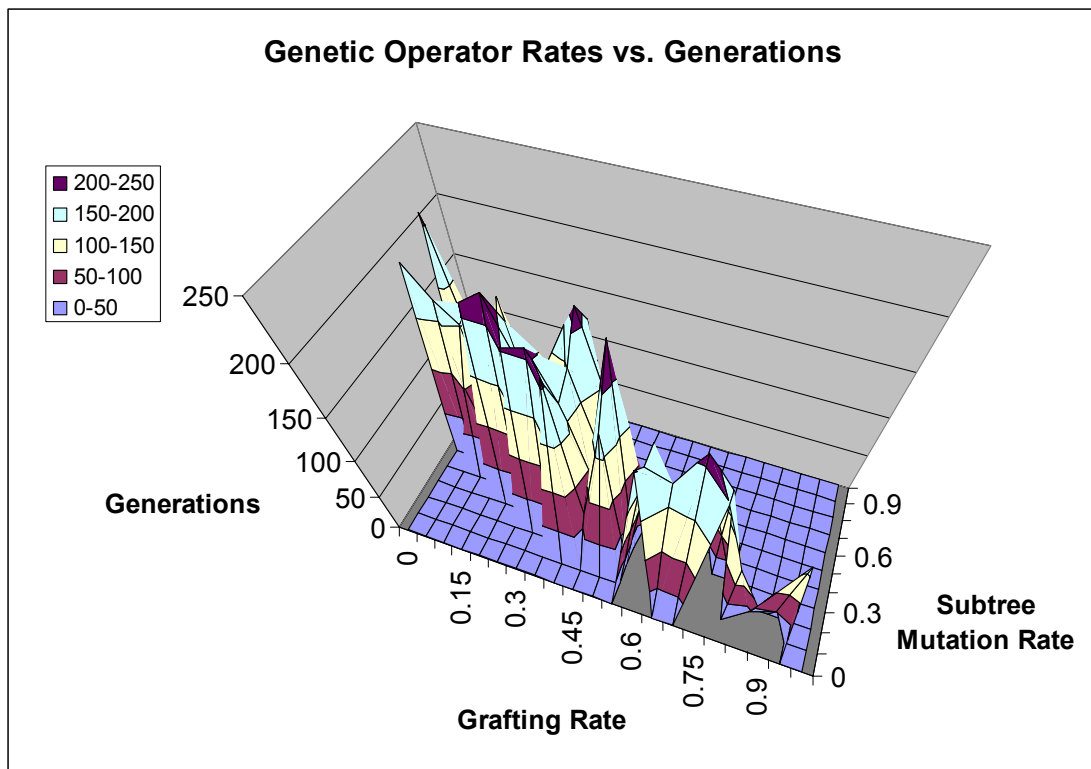


Figure 2 – fi06 dataset

It is shown here, in Figure 2, that the best values for the three operators are approximately 0.4 for subtree mutation, 0.40 for node mutation, and 0.2 for grafting. This is where there is a great deal of clustering of values which reach the global optimum in very few generations.

It appears that higher rates of node mutation and subtree mutation produce better results than higher rates of grafting. This makes sense in terms of speciation and local versus global searching. A node mutation has a higher chance of making a small change to the phenotype of a tree due to the high number of nodes that occur in non-critical leaf regions. Subtree mutation also has a good chance of effecting local search as the chance that smaller branches will be swapped is greater than that of larger branches. This is because there are a higher number of smaller branches than of larger ones. Grafting tends to be a global search operator because in order to graft a branch from one tree onto another, the structure of the recipient tree can be modified greatly, producing an entirely different species, and thereby searching a different area of the solution space.

Calibration Test 2

Test 2 was carried out on the standard data set abz6. This is a 10 machine, 10 job problem, so rather than each run lasting just 200 generations, this test was set for 500 generations. Due to the high number of generations and the large size of the problem, each run within this set, i.e. each combination of

genetic operator rates, took up to 1 hour with 50 trees per sample; the same as for test 1. As for test 1, this only allowed 2 runs per combination of operator rates to be completed, though the interval for node mutation was expanded from 0 to 0.4 up to 0 to 0.8.

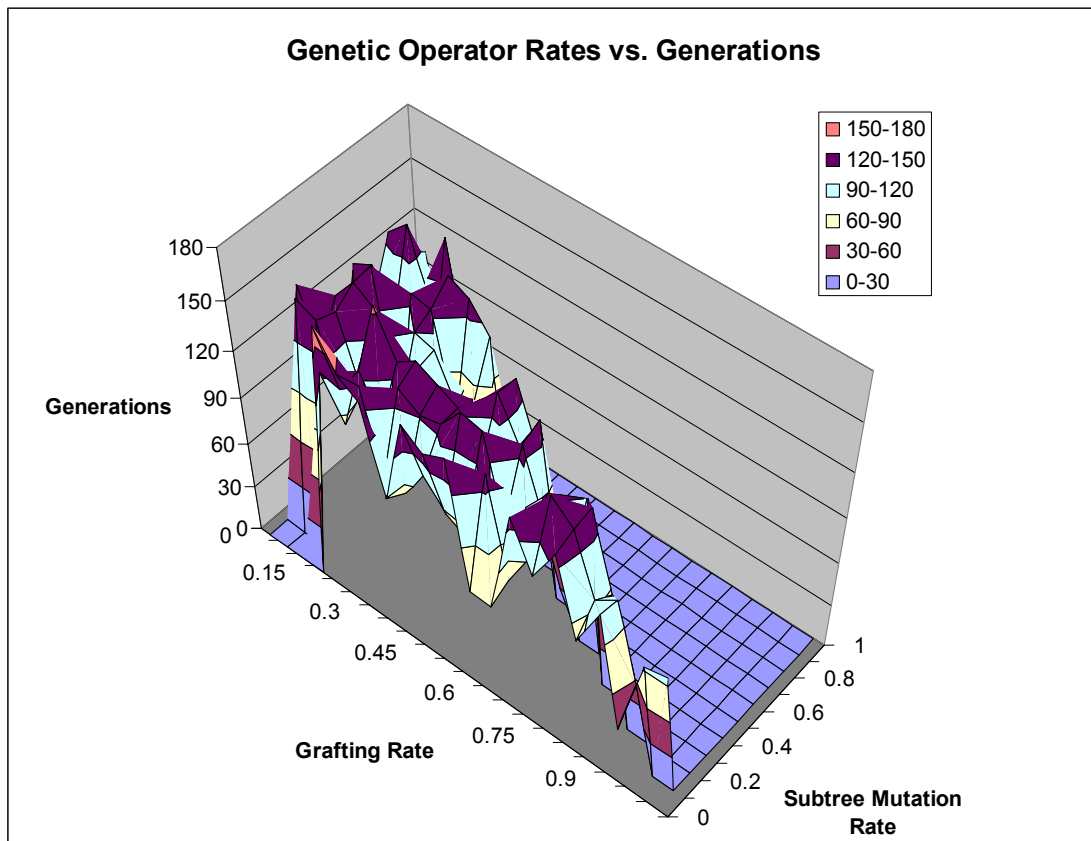


Figure 3 – abz6 dataset

The style of results as shown in Figure 3 is different than in Figure 2. In this case, the best result is not known, but the best result found, according to (Blum, 02; Ombuki and Ventresca, 02) is 943. Because of this lack of knowledge as to a definite best result, the values plotted in Figure 3 are each result subtracted from the worst result. This gives the same effect as in Test 1 of showing the best results as peaks instead of troughs.

During this test, scores as low as 923 were achieved which is a major improvement over the known best of 943, so this test set will be revisited in more detail in the results section.

As can be seen from the graph, the best combination of genetic operators appears to cover the region corresponding where node mutation is 0.75. The other values along this line that appear to be in the middle of the high-scoring band are 0.2 for grafting and 0.05 for subtree mutation. These values corroborate the results from Test 1 in that the weighting appears to be about 0.8 for local search operators and 0.2 for global search. The rate of 0.2 for grafting will be used from here on as the appropriate rate for global searching and, as there is no

other data at this point, the rate of node mutation will vary from 0.4 up to 0.75 depending upon the size of the experiment being run. The rate of subtree mutation will, of course, be constrained by the choices for the other two operators.

4.2 Weighting

Using the weighting scheme described in 3.1.1, the notion was developed that decreasing the strength of the weighting as generations increased could actually be useful to improving the quality of results. The strength of the weighting can be considered to be analogous to an evolutionary pressure, or the advantage that a fit individual will have over an unfit individual. Due to this, it could be seen as counter-intuitive that decreasing the evolutionary pressure would produce better individual chromosomes, but a possible explanation of this theory is shown in Figure 4, below, with an accompanying explanation.

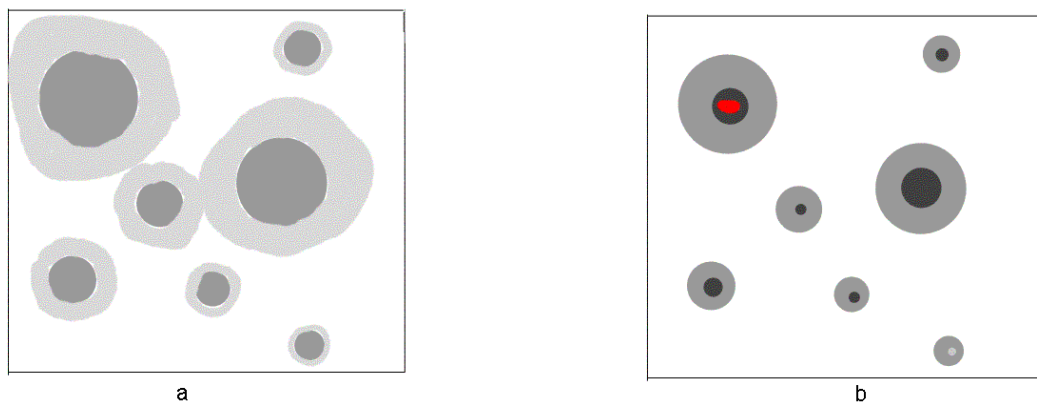


Figure 4

Near the start of a test with an evolutionary algorithm, the evolutionary pressure is high, but the average score is bad. This is described by part a of Figure 4. The dark grey areas represent parts of the solution space surrounding local optima. These are the areas into which the algorithm is guiding the chromosomes. Not all of them will be found, however, and the light grey areas surrounding the dark grey areas represent the parts of the solution space that will eventually lead into one of the optima. These light grey areas will only lead into the optima because the current average scores are so bad, that even with high pressure, much of the solution space is reachable.

After a while, part b of Figure 4 is reached. Good best and average scores will reduce the size of the areas which will lead to a local optimum. The black regions indicate these optima. The patch inside the black region near the top-left represents a current chromosome stuck in a local optimum that is not the global optimum. Near the bottom left is a light patch inside a dark grey region. This light patch represents the global optimum in the search space. There is very little chance that a chromosome in the local optimum will manage to mutate and hit the global optimum which is very small. This is why, after some generations have passed, if the evolutionary pressure is eased off, the dark grey region of part

b of Figure 4 can be reached, instead of just the pinhead in the middle of it. This does not guarantee that a better result will be reached, but it increases the chances of it happening.

This is why the weighting aggression should be lowered, so that the light grey areas can be reached. This way, it still restricts results to being far better than the earliest generations, but it allows larger deviations from the current best so that the other optima can be found.

This is analogous to human evolution. In the animal kingdom, the weak are killed with no regard for whether they could become stronger in a different way if given the opportunity. Humanity, however, has evolved to the level where the weak are allowed to survive and breed. This is a trade-off. It allows those who are weak with minimal chance of imparting anything useful to society to exist for the benefit of allowing the chance few, who manage to overcome their lack of traditional fitness to gain fitness in other ways.

For example, the humble nerd would have begun in human evolution as the creature unable to hunt effectively due to lacklustre hand-eye coordination, small muscles and a constant wheezing that would scare off possible quarry. The nerd however, managed to utilise his spare time to other ends, such as inventing tools and discovering ideas that were of more benefit to his society than any individual's muscles.

So basically, it is a matter of being strict until a viable result is achieved, and then relaxing until the optimal result is reached. The time to switch is possibly located at the point of diminishing returns, such as where no significant improvement has been noticed in x generations.

This weighting scheme would result in the best score graph changing from an exponential graph asymptoting at an optimal result to a series of exponentials, each smaller than the previous, representing the finding of new local optima, and all combining to make an approximate exponential graph, overall. The average score graph would then be expected to increase briefly after each new local optimum is found, as in the first generations after an optimum is found, the genetic operators would produce more poor results than after the new optimum becomes the standard.

4.3 Experimental Procedure

The following experiments were run with genetic operator rates of 0.75, 0.05 and 0.2 for node mutation, subtree mutation and grafting, respectively and sample sizes of 100. Each experiment was made up of two parts. Part *i* was run for 1500 generations with values of a and b set as 5 and 1, respectively. Part *ii* of each experiment was run for 1500 generations divided into blocks of 300 generations where the values of a and b ranged linearly from 5 and 1, respectively, to 9 and 17. This was in order to test the theory that lowering the evolutionary pressure could lead to better long term results.

For each of these experiments, 3 separate runs were conducted. This was done in order to determine a reasonable average result that had statistical significance. Results are shown in comparison to the ACO and TS algorithms

defined in (Blum, 02) and the LSGA and Hybrid algorithms defined in (Ombuki and Ventresca, 02).

5 Experimental Results and Discussion

5.1 Experiment 1 - abz6 dataset

As can be seen from Table 1, the JMSC algorithm has produced a result about 1.3% more efficient than the best known result found. This confirms the algorithms ability to surpass other algorithms in at least one area and possibly demonstrates that the results of 932 and 923, which were found during sensitivity analysis are local optima in the search space using this method. Unfortunately, in none of these experiments was the 923 result replicated, though this would seem to be a matter of chance to some extent and the result would probably be found given more runs.

Exp	Best known	ACO	TS	JMSC	
		Best	Best	Best	Average
i	943	947	943	952	979.3
ii	943	947	943	932	980.3

Table 1

Another result which can be implied from Table 1 is that method *ii* does indeed provide better results than method *i*, and it can be seen that the method does allow weaker results to exist as well. Both methods have almost identical averages, yet with the best results differing by 20, it means that the other 2 results for method *ii* are on average 10 worse, each, than their method *i* counterparts.

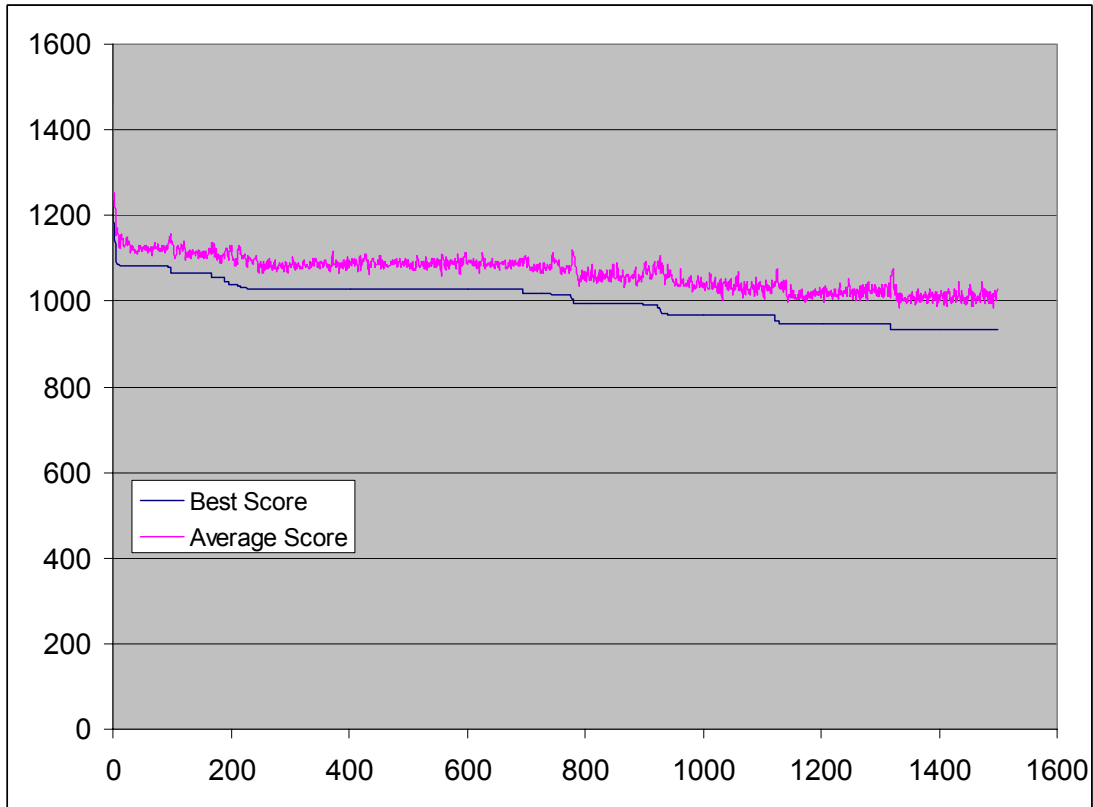


Figure 5

Figure 5 shows the best and average scores, or makespans, plotted against generation number.

As can be seen in this figure, there is a series of changes in the best score line that each look like exponentials, and each little exponential seems to have a smaller effect than the previous one, with one exception around the 700th generation mark. Slightly after each of these decreases in the makespan, there is a brief increase in the average makespan line. This is exactly what was expected. What is happening is that the increased ability for poor chromosomes to breed occasionally allows a poor chromosome to produce the best chromosome to date. This allows many poorer chromosomes to be produced when the new best individual becomes the template from which a new local optimum is explored.

5.2 Experiment 2 – la24 dataset

The la24 dataset defines a 15 job, 10 machine problem. It is considered to be a tough benchmark (Blum, 02). This opinion is justified considering the results in Table 2. It can be seen that part *ii* of the experiment exceeds the performance of part *i* by a small amount, but it trails a fair distance behind the other methods in the comparison.

Exp	Best known	ACO	TS	LSGA	Hybrid	JMISC	
		Best	Best	Best	Best	Best	Average
i	935	943	939	1032	1001	1070	1087

ii	935	943	939	1032	1001	1064	1076.7
----	-----	-----	-----	------	------	------	--------

Table 2

As can be seen in Figure 6, which represents the best and average scores for the best run for part *ii* of the la24 dataset, the main problem is clearly that JMSC was unable to locate any better local optima after the first relaxation of evolutionary pressure. This would imply that perhaps 300 generations was too long to wait if reducing the pressure were to produce a significant improvement. It could be that for a GTA designed in this method, there are just no better optima within easy reach of the most easily accessible local optima.

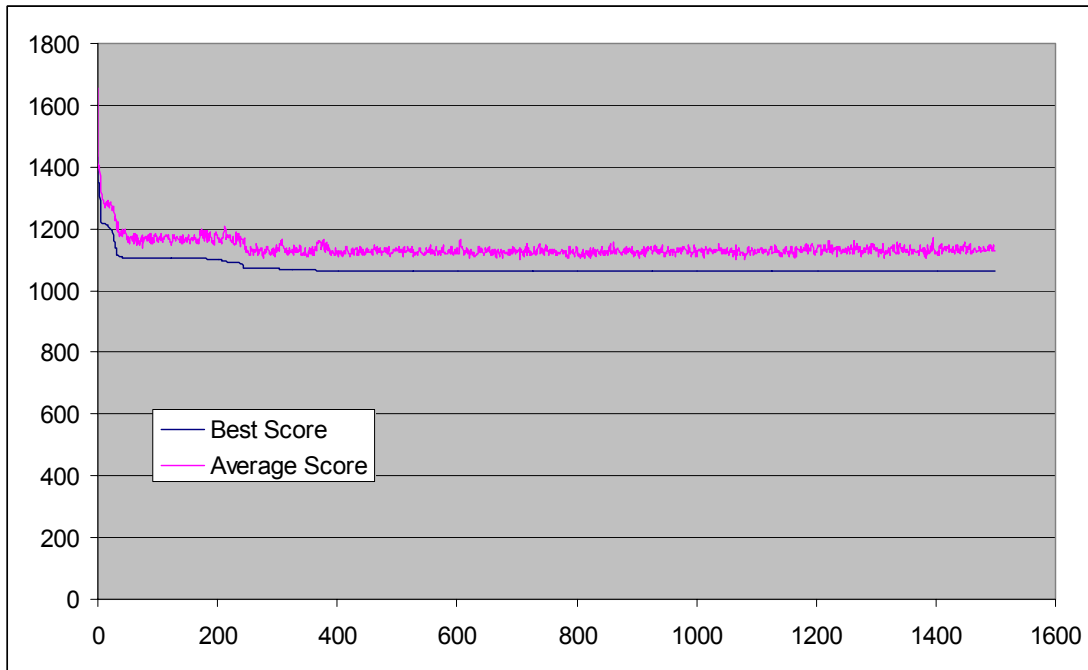


Figure 6

5.3 Experiment 3 – la25 dataset

The la25 dataset also defines a 15 job, 10 machine problem and is considered to be a tough benchmark (Blum, 02). The results as shown in Table 3 are equally as unimpressive as those shown in Table 2.

Exp	Best known	ACO	TS	LSGA	Hybrid	JMSC	
		Best	Best	Best	Best	Best	Average
i	977	978	977	1047	1031	1113	1120.3
ii	977	978	977	1047	1031	1107	1119.7

Table 3

While the results in Table 3 show only ACO, TS LSGA and Hybrid vs. JMSC, there are also other methods detailed in (Binato *et al.*, 00; Cheng and Smith, 95) that are comparable to these four methods, meaning that they surpass JMSC in quality of results.

Yet again, according to Figure 7 which depicts the best and average makespans from the best run in part *ii*, it seems that waiting 300 generations to

decrease the evolutionary pressure is too long. If time had permitted, more than 3 runs per test would have been conducted which, while not lowering the average much, might have found a better “best” result. Also, it seems that experiments 2 and 3 would have benefited from decreasing the evolutionary pressure to prevent stagnation at much smaller intervals.

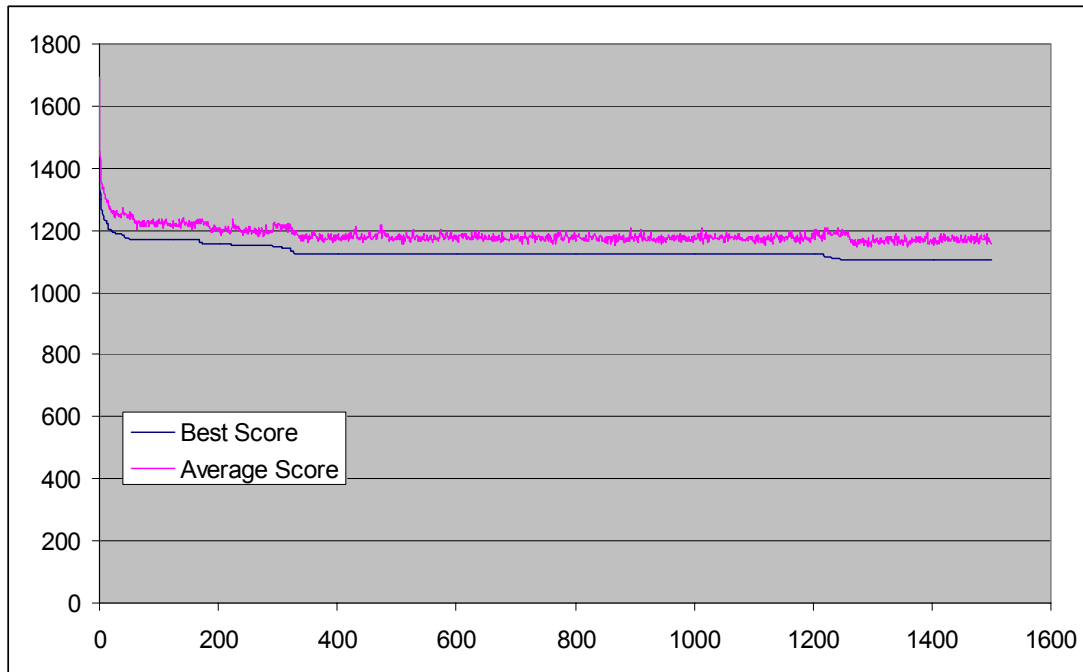


Figure 7

5.4 Experiment 4 – ft06 dataset

The ft06 dataset is a 6 job, 6 machine dataset. This experiment is unlike the 3 experiments shown above in that it was not difficult at all to surpass the best known result. In fact, it was surpassed in about 75% of cases during the sensitivity analysis and the main question was about how many generations it would take.

Figure 2 shows how many generations on average it took for each of the combination of genetic operator rates to reach 54, subtracted from 250. All of the points on the graph that are greater than zero on the z axis reached the optimal value, and of the points that are zero, where ever the node mutation rate would be grater than 0.4, the experiment was not tried.

6 Conclusion

As it was shown in the results, the abz6 dataset benefited greatly from adjusting the evolutionary pressure after every 300 generations. It allowed the JMSC algorithm to produce the best known results, rather than below average results when compared to other methods. Other datasets, such as the la24 and la25 datasets did not seem to benefit much and the results for these datasets were found lacking when compared to other methods, including (Binato *et al.*, 00;

Cheng and Smith, 95). It seems to be that the search mechanism became entrenched in searching just one or two local optima. From looking at the results, it seems that if the evolutionary pressure were decreased more frequently than every 300 generations, maybe as often as every 50 generations, the algorithm could be kept out of the trap of a few very attractive, but ultimately fruitless optima. It would seem that rather than setting a fixed number for the trigger in reducing the evolutionary pressure, there should be another

7 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

Appendix A – Main Program Code

Note: Because this program was written in Delphi, all of the visual aspects are missing, including the parts that link event handlers to events.

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, JvScrollBar, StdCtrls,
  JvComboBox, JvxCtrls, JvSpin, JvButton, OleServer,
  MSXML2_TLB, Math, ExtCtrls, JvPanel, JvWinDialogs,
  JvCheckBox, JvGIF, JvImage;

type
  TShopForm = class(TForm)
    ItemsBox: TJvScrollBar;
    JvxLabel3: TJvxLabel;
    JvxLabel4: TJvxLabel;
    OutputPanel: TJvScrollBar;
    SetupBox: TGroupBox;
    ActionsBox: TGroupBox;
    JvxLabel1: TJvxLabel;
    JvxLabel2: TJvxLabel;
    JvxLabel5: TJvxLabel;
    MachinesBox: TJvxSpinEdit;
    JobsBox: TJvxSpinEdit;
    ChangeItems: TJvButton;
    SampleSize: TJvxSpinEdit;
    MakeGen0Button: TButton;
    ScoreSolutions: TButton;
    DrawBest: TJvButton;
    Mutate: TJvButton;
    NodeMutationRate: TJvxSpinEdit;
    JvxLabel6: TJvxLabel;
    Bevel1: TBevel;
    JvxLabel7: TJvxLabel;
    SubMutationRate: TJvxSpinEdit;
    JvxLabel8: TJvxLabel;
    GraftRate: TJvxSpinEdit;
    SaveCSV: TJvButton;
    DoXGens: TJvButton;
    GensToDo: TJvxSpinEdit;
    JvxLabel12: TJvxLabel;
```

```

CloneRate: TJvxSpinEdit;
LoadButton: TJvButton;
OpenDialog: TJvOpenDialog2000;
StoreGens: TJvCheckBox;
ScaleValue: TJvxSpinEdit;
JvxLabel9: TJvxLabel;
JvImage1: TJvImage;
JvImage2: TJvImage;
NormValue: TJvxSpinEdit;
JvxLabel10: TJvxLabel;
TestFileButton: TJvButton;
runtestsbutton: TJvButton;
procedure ChangeItemsClick(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure MakeGen0ButtonClick(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure ScoreSolutionsClick(Sender: TObject);
procedure DrawBestClick(Sender: TObject);
procedure FormResize(Sender: TObject);
procedure MutateClick(Sender: TObject);
procedure MutationRateChange(Sender: TObject);
procedure SaveCSVClick(Sender: TObject);
procedure DoXGensClick(Sender: TObject);
procedure GensToDoChange(Sender: TObject);
procedure LoadButtonClick(Sender: TObject);
procedure TestFileButtonClick(Sender: TObject);
procedure runtestsbuttonClick(Sender: TObject);
private
    Private declarations
    Solutions : TDOMDocument;
    Solution : IXMLDOMEElement;

    AllGens : TDOMDocument;
    LastGen : IXMLDOMEElement;
    GenNum : Integer;
    BestScores : Array of Single;
    AverageScores : Array of Single;

    tableInc : Integer;
    Scores : Array of Array[0..2] of Single;
    BestScoresGen : Array of Single;
    MinScore : Single; // the score for the best
tree in the sample, where lower is better.
    BestTree : Integer; // the index of the tree with
the lowest score.
    BestTreeJobs : Array of Array of Single;

```

```

BestTreeMacs : Array of Array of Single;

public
    Public declarations

    Items      : Array of Array of Array[0..1] of
TJvxSpinEdit; // first element of 2 element part is
time, second element is order position
    JobLabels  : Array of TJvxLabel;
    MachineLabels : Array of TJvxLabel;
    runNumber  : Integer;
    procedure reverse(var arr : array of Integer);
    function  createRandomTree(dir : string; var
number : Array of Integer) : IXMLDOMEElement;
    procedure getItems(root : IXMLDOMEElement; var
table : array of Integer; traversal : Integer);
    function  deleteItems(root : IXMLDOMEElement; var
donorNodeList : array of Integer) : IXMLDOMEElement;
    procedure sortJobs(var jobs : Array of single);
    function  getNode(Element : IXMLDOMEElement; var
number : Integer) : IXMLDOMEElement;
    function  getNodeByNumber(root : IXMLDOMEElement;
number : Integer) : IXMLDOMEElement;
    function  canNodeBeDeleted(Element :
IXMLDOMEElement; number : Integer) : boolean;
    function  isRelated(x, y : IXMLDOMEElement) :
Boolean;
    function  isAncestor(x, y : IXMLDOMEElement) :
Boolean;
    function  graft(tree, branch : IXMLDOMEElement) :
Boolean;
    end;

var
    ShopForm: TShopForm;
    ticks_per_second : TLargeInteger;
    Filename : string;
    start_time, end_time : TLargeInteger;

implementation

    $R *.DFM

    procedure TShopForm.ChangeItemsClick(Sender: TObject);
var
    i, j      : Integer;

```

```

Item      : TjvxSpinEdit;
ItemLabel : TjvxLabel;
OldItemsLength, OldItemsiLength : Integer;
begin
  FormCreate(ChangeItems);
  for i := 0 to high(Items) do begin
    for j := 0 to high(Items[i]) do begin
      if (i >= MachinesBox.AsInteger) or (j >=
JobsBox.AsInteger) then begin
        FreeAndNil(Items[i][j][0]);
        FreeAndNil(Items[i][j][1]);
      end;
    end;

    if i = 0 then begin
      OldItemsiLength := Length(Items[0]);
    end;
  end;
  OldItemsLength := Length(Items);
  setLength(Items, MachinesBox.AsInteger,
JobsBox.AsInteger);
  for i := 0 to MachinesBox.AsInteger - 1 do begin
    for j := 0 to JobsBox.AsInteger - 1 do begin
      if (i >= OldItemsLength) or (j >=
OldItemsiLength) then begin
        Items[i][j][0] :=
TJvxSpinEdit.Create(ItemsBox);
        with Items[i][j][0] do begin
          Width      := 55;
          Decimal    := 1;
          ValueType  := vtFloat;
          MinValue   := 0;
          Top        := 30 + i * (Height + 5);
          Left       := 80 + j * (Width + 60);
          Parent     := ItemsBox;
          Value      := trunc(Random(20)) + 1;
        end;
        Items[i][j][1] :=
TJvxSpinEdit.Create(ItemsBox);
        with Items[i][j][1] do begin
          Width      := 45;
          Decimal    := 0;
          ValueType  := vtInteger;
          MinValue   := 0;
          Top        := 30 + i * (Height + 5);

```

```

        Left      := Items[i][j][0].Left +
Items[i][j][0].Width + 1;
        Parent    := ItemsBox;
        Value     := 0;
    end;
end;
end;
end;

end;

procedure TShopForm.FormCreate(Sender: TObject);
var
    Time : TLabel;
    i     : Integer;
begin
    ItemsBox.DoubleBuffered := True;
    Randomize;
    if Sender <> ChangeItems then
        Filename := '';
        FreeAndNil(AllGens);
        AllGens := TDOMDocument.Create(Self);
        AllGens.DefaultInterface.documentElement :=
AllGens.DefaultInterface.createElement('generations');

AllGens.DefaultInterface.insertBefore(AllGens.DefaultI
nterface.createElement('xml',
'version="1.0" encoding="UTF-8"'),
AllGens.DefaultInterface.documentElement);

    for i := 0 to 300 do begin
        Time := TLabel.Create(OutputPanel);
        with Time do begin
            Caption := IntToStr(i * 2);
            Left := (50 + i * 25) - Width div 2;
            Top := 10;
            Parent := OutputPanel;
        end;
    end;
    GenNum := 0;
    GensToDoChange(Self);
end;

procedure TShopForm.MakeGen0ButtonClick(Sender:
TObject);
var

```

```

ItemArray : Array of Integer;
h, i, j, k : Integer;
Temp      : Integer;
Duplicate : Boolean;
begin
  Randomize;
  FreeAndNil(AllGens);
  AllGens := TDOMDocument.Create(Self);
  AllGens.DefaultInterface.documentElement :=
AllGens.DefaultInterface.createElement('generations');

AllGens.DefaultInterface.insertBefore(AllGens.DefaultI
nterface.createElement('xml',
'version="1.0" encoding="UTF-8"'),
AllGens.DefaultInterface.documentElement);

  FreeAndNil(Solutions);
  Solutions := TDOMDocument.Create(Self);
  Solutions.DefaultInterface.documentElement :=
Solutions.DefaultInterface.createElement('solutions');

Solutions.DefaultInterface.insertBefore(Solutions.Defa
ultInterface.createElement('xml',
'version="1.0" encoding="UTF-8"'),
Solutions.DefaultInterface.documentElement);

  Setlength(ItemArray, MachinesBox.AsInteger *
JobsBox.AsInteger);

  for j := 0 to SampleSize.AsInteger - 1 do begin
    for h := 0 to High(ItemArray) do
      ItemArray[h] := 0;
      for h := 0 to High(ItemArray) do begin
        repeat
          Duplicate := False;
          Temp := Round(Random(Length(ItemArray)));
          for i := 0 to h - 1 do begin
            if Temp = ItemArray[i] then begin
              Duplicate := True;
              break;
            end;
          end;
        until not Duplicate;
        ItemArray[h] := Temp;
      end;
      Solution := createRandomTree('top', ItemArray);
    end;
  end;
end;

```

```

Solutions.DefaultInterface.documentElement.appendChild
(Solution);
end;
try
    Solutions.DefaultInterface.save('.\tree.xml');
except
end;

    GenNum := 1;

end;

function TShopForm.getNodeByNumber(root :
IXMLDOMEElement; number : Integer) : IXMLDOMEElement;
var
    // gets the node with the given number
    i : Integer;
begin
    Result := nil;
    if
strToInt(root.attributes.getNamedItem('number').text)
= number then begin
        Result := root;
        exit;
    end;
    for i := 0 to root.childNodes.length - 1 do begin
        Result :=
getNodeByNumber(IXMLDOMEElement(root.childNodes[i]),
number);
        if Result <> nil then exit;
    end;
end;

function TShopForm.createRandomTree(dir : string; var
number : Array of Integer) : IXMLDOMEElement;
var
    numberCopy    : array of Integer;
    list1, list2  : array of Integer;
    i              : Integer;
    Att            : IXMLDOMAttribute;
begin
    Result := nil;
    if length(number) = 0 then
        exit;
    setLength(numberCopy, high(number));
    setLength(list1, high(number));

```

```

    setLength(list2, high(number));
    for i := 0 to high(numberCopy) do begin
        numberCopy[i] := number[i + 1];
        list1[i] := number[i + 1];
        list2[i] := number[i + 1];
    end;
    setlength(list1, ceil(length(numberCopy) / 2));
    reverse(list2);
    setlength(list2, length(numberCopy) div 2);
    Result :=
Solutions.DefaultInterface.createElement('node');
    Result.setAttribute('number', number[0]);
    Result.setAttribute('dir', dir);
    if length(list1) > 0 then begin
        Result.appendChild(createRandomTree('left',
list1));
    end else
        exit;
    if length(list2) > 0 then begin
        Result.appendChild(createRandomTree('right',
list2));
    end else
        exit;
end;

procedure TShopForm.reverse(var arr : array of
Integer);
var
    Temp : Integer;
    i : Integer;
begin
    for i := 0 to (length(arr) div 2) - 1 do begin
        Temp := arr[i];
        arr[i] := arr[high(arr) - i];
        arr[high(arr) - i] := Temp;
    end;
end;

procedure TShopForm.FormDestroy(Sender: TObject);
begin
    Solutions.free;
end;

procedure TShopForm.ScoreSolutionsClick(Sender:
TObject);
var

```

```

h, i, j, k, p      : Integer;
TopElement        : IXMLDOMEElement;

table             : Array of Array of Integer;
tableId          : Array of Integer;
MachineTimes     : Array of Array[0..2] of Array of
Single; // stores sorted start-end times for
production on each machine. should always increase,
so it should be automatically sorted
  JobTimes       : Array of Array[0..2] of Array of
Single; // stores sorted start-end times for
production of each job. It is possible to insert items
into gaps here
  TasksProcessed : Array of Integer;           // how many
tasks have been processed for each job

// OutputTableFile : TFileStream;
// OutputTable     : String;

MacNum           : Integer;
MacTime          : Single;
JobNum           : Integer;
JobTime          : Single;
OrderTime        : Single;
GapStart         : Single;
StartTime        : Single;

tableCopy1       : Array of Integer;
tableCopy2       : Array of Integer;
begin
  MinScore := -1;
  Setlength(BestScores, GenNum);
  Setlength(AverageScores, GenNum);
  BestScores[high(BestScores)] := -1;
  AverageScores[high(AverageScores)] := 0;
  setLength(table, MachinesBox.AsInteger,
JobsBox.AsInteger);
  setLength(tableId, MachinesBox.AsInteger *
JobsBox.AsInteger);

  setLength(MachineTimes, MachinesBox.AsInteger);
  setLength(JobTimes, JobsBox.AsInteger); // for each
Job, for each traversal, JobTimes stores the start and
end times for when the job has been machined
  setLength(Scores, SampleSize.AsInteger);

```

```

    setLength(BestScoresGen, SampleSize.AsInteger);
// OutputTable :=
'<html><head><title>tables</title></head><body>';

    setLength(TasksProcessed, JobsBox.AsInteger); //
keeps track of the highest task number processed so
far for each Job
    for i := 0 to SampleSize.AsInteger - 1 do begin
        for p := 0 to high(scores[i]) do begin
            for j := 0 to high(MachineTimes) do
                setLength(MachineTimes[j][p], 0);
            for j := 0 to high(JobTimes) do
                setLength(JobTimes[j][p], 0);
            Scores[i][p] := -1; // initialised
to -1 so that any time will be bigger;
        end;
        for p := 0 to high(Scores[i]) do begin
            TopElement :=
IXMLDOMElement(Solutions.DefaultInterface.documentElem
ent.childNodes[i]);
            tableInc := 0;
            getItems(TopElement, tableId, p);
            setLength(tableCopy1, length(tableId));
            for j := 0 to high(tableCopy1) do
                tableCopy1[j] := tableId[j]; //
initialising the two copies of tableId.
            for j := 0 to high(TasksProcessed) do
                TasksProcessed[j] := -1;
            while length(tableCopy1) > 0 do begin
                setLength(TableCopy2, 0);
                for j := 0 to high(tableCopy1) do begin
//                    find the machine number
                    MacNum := tableCopy1[j] div
JobsBox.AsInteger;
//                    find the job number
                    JobNum := tableCopy1[j] mod
JobsBox.AsInteger;
//                    only process the node if all previous tasks
for the job have been done
                    if Items[MacNum][JobNum][1].AsInteger =
TasksProcessed[JobNum] + 1 then begin
                        TasksProcessed[JobNum] :=
Items[MacNum][JobNum][1].AsInteger;
//                        get earliest available time to maintain
the ordering
                        if TasksProcessed[JobNum] = 0 then

```

```

        OrderTime := 0
    else begin
        OrderTime :=
JobTimes[JobNum][p][high(JobTimes[JobNum][p])];
    end;
//    get earliest Available time for machine
    try
        MacTime :=
MachineTimes[MacNum][p][high(MachineTimes[MacNum][p])]
;
    except
        MacTime := 0;
    end;
//    get first gap in job processing starting
>= MacTime

    try
        k := 0;
        while k < length(JobTimes[JobNum][p]) do
begin
            if k > 0 then begin
                if (JobTimes[JobNum][p][k] -
JobTimes[JobNum][p][k - 1] >=
Items[MacNum][JobNum][0].Value) then begin
                    if JobTimes[JobNum][p][k - 1] >=
MacTime then begin
                        GapStart :=
JobTimes[JobNum][p][(k div 2) - 1];
                        break;
                    end;
                end;
            end;
            k := k + 2;
        end;
        if k = length(JobTimes[JobNum][p]) then
            JobTime :=
JobTimes[JobNum][p][high(JobTimes[JobNum][p])]
        else
            JobTime := GapStart;
    except
        JobTime := 0;
    end;
    StartTime := Max(JobTime, MacTime);
    StartTime := Max(StartTime, OrderTime);
//    Allocate memory to store times and
processes

```

```

        setlength(MachineTimes[MacNum][p],
length(MachineTimes[MacNum][p]) + 3);
        setlength(JobTimes[JobNum][p],
length(JobTimes[JobNum][p]) + 2);
//          store times

MachineTimes[MacNum][p][high(MachineTimes[MacNum][p])
- 2] := MacNum * JobsBox.AsInteger + JobNum;

MachineTimes[MacNum][p][high(MachineTimes[MacNum][p])
- 1] := StartTime;

MachineTimes[MacNum][p][high(MachineTimes[MacNum][p])]
:= StartTime + Items[MacNum][JobNum][0].Value;

JobTimes[JobNum][p][high(JobTimes[JobNum][p]) - 1] :=
StartTime;

JobTimes[JobNum][p][high(JobTimes[JobNum][p])] :=
StartTime + Items[MacNum][JobNum][0].Value;

//          find the highest time in MachineTimes as
the fitness of the current tree
        Scores[i][p] := Max(Scores[i][p],
MachineTimes[MacNum][p][high(MachineTimes[MacNum][p])]
);

//          sort and store job times
        sortJobs(JobTimes[JobNum][p]);
        end else begin // if this node cannot be
processed yet, because previous tasks in the job
haven't been done yet
            setLength(TableCopy2, length(TableCopy2) +
1);
                TableCopy2[high(tableCopy2)] :=
TableCopy1[j];
            end;
        end;
        setLength(tableCopy1, length(tableCopy2));
        for k := 0 to high(tableCopy2) do
            tableCopy1[k] := tableCopy2[k];
        end;
    end;
// now that times have been determined, do some
output

```

```

    OutputTable := OutputTable + '<table>'#13#10;
    OutputTable := OutputTable +
'<thead>'#13#10'<tr>'#13#10'<td>&nbsp;</td>'#13#10;
    for j := 1 to (2 * JobsBox.AsInteger) do begin
        if j mod 2 = 1 then
            OutputTable := OutputTable + '<th
scope="col">Start Time</th>'#13#10
        else
            OutputTable := OutputTable + '<th
scope="col">End Time</th>'#13#10;
        end;
        OutputTable := OutputTable +
'</tr>'#13#10'</thead>'#13#10;
        for j := 0 to MachinesBox.AsInteger - 1 do begin
            OutputTable := OutputTable + '<tr>'#13#10'<th
scope="row">Machine ' + IntToStr(j) + '</th>'#13#10;
            for k := 0 to (3 * JobsBox.AsInteger) - 1 do
begin
                if k mod 3 <> 0 then
                    OutputTable := OutputTable + '<td>' +
FloatToStr(MachineTimes[j][k]) + '</td>'#13#10;
                end;
            OutputTable := OutputTable + '</tr>'#13#10;
        end;
        OutputTable := OutputTable + '</table>'#13#10;

        for k := 0 to SampleSize.AsInteger - 1 do begin
            BestScoresGen[k] := Scores[k][0];
            for j := 1 to high(Scores[k]) do begin
                if Scores[k][j] < BestScoresGen[k] then
                    BestScoresGen[k] := Scores[k][j];
            end;
        end;

        AverageScores[GenNum - 1] := AverageScores[GenNum
- 1] + BestScoresGen[i] / SampleSize.Value;
        if (MinScore > BestScoresGen[i]) or (MinScore < 0)
then begin
            MinScore := BestScoresGen[i];
            BestScores[GenNum - 1] := MinScore;
            BestTree := i;

            p := 0; // get traversal of best score for
current tree
            for k := 1 to high(scores[i]) do begin
                if Scores[i][k] < Scores[i][p] then

```

```

        p := k;
    end;

    setLength(BestTreeJobs, Length(JobTimes));
    setLength(BestTreeMacs, Length(MachineTimes));
    for k := 0 to high(BestTreeJobs) do begin
        setlength(BestTreeJobs[k],
Length(JobTimes[k][p]));
        for j := 0 to high(BestTreeJobs[k]) do begin
            BestTreeJobs[k][j] := JobTimes[k][p][j];
        end;
    end;
    for k := 0 to high(BestTreeMacs) do begin
        setlength(BestTreeMacs[k],
Length(MachineTimes[k][p]));
        for j := 0 to high(BestTreeMacs[k]) do begin
            BestTreeMacs[k][j] := MachineTimes[k][p][j];
        end;
    end;
end;
end;

try
    OutputTableFile :=
TFileStream.Create('\outputTable.html', fmCreate or
fmOpenWrite);

OutputTableFile.Write(PChar(OutputTable)^, Length(Outpu
tTable));
except
    MessageDlg('Could not create the output
file.'#13#10'Please check that there is enough disk
space and that the disk is not write-protected.',
mtError, [mbOK], 0);
end;
FreeAndNil(OutputTableFile);

end;

procedure TShopForm.getItems(root : IXMLDOMEElement;
var table : array of Integer; traversal : Integer);
var
    i : Integer;
begin
    case traversal of

```

```

    0: begin
// prefix traversal
    table[tableInc] :=
StrToInt(root.attributes.getNamedItem('number').text);
    tableInc := tableInc + 1;
    for i := 0 to root.childNodes.length - 1 do
        getItems (IXMLDOMEElement (root.childNodes[i]),
table, traversal);
    end;
    1: begin
// infix traversal
    if root.childNodes.length >= 1 then begin
        if
root.firstChild.attributes.getNamedItem('dir').text =
'left' then begin
            getItems (IXMLDOMEElement (root.firstChild),
table, traversal);
        end;
    end;
    table[tableInc] :=
StrToInt(root.attributes.getNamedItem('number').text);
    tableInc := tableInc + 1;
    if root.childNodes.length >= 1 then begin
        if
root.lastChild.attributes.getNamedItem('dir').text =
'right' then begin
            getItems (IXMLDOMEElement (root.lastChild),
table, traversal);
        end;
    end;
    end;
    2: begin
// postfix traversal
    for i := 0 to root.childNodes.length - 1 do
        getItems (IXMLDOMEElement (root.childNodes[i]),
table, traversal);
    table[tableInc] :=
StrToInt(root.attributes.getNamedItem('number').text);
    tableInc := tableInc + 1;
    end;
end;
end;

procedure TShopForm.sortJobs (var jobs : Array of
single);
var

```

```

    i : Integer;
    TempJobTime      : Array[0..1] of Single;
begin
    for i := 2 to high(jobs) do begin
        if jobs[i] < jobs[i - 1] then begin
            TempJobTime[0] := jobs[i];
            TempJobTime[1] := jobs[i + 1];
            jobs[i]         := jobs[i - 2];
            jobs[i + 1]     := jobs[i - 1];
            jobs[i - 2]     := TempJobTime[0];
            jobs[i - 1]     := TempJobTime[1];
        end;
    end;
end;

procedure TShopForm.DrawBestClick(Sender: TObject);
var
    i, j          : Integer;
    StartTime, EndTime : Single;
    Shape         : TShape;
    ProcessNum    : TLabel;
begin
    for i := 0 to outputPanel.ControlCount - 1 do begin
        try
            TShape(outputPanel.Controls[i]).brush.Bitmap :=
nil; // trying to cause an exception if it is not a
shape
            outputPanel.Controls[i].free;
        except
            end;
        try
            if TLabel(outputPanel.Controls[i]).tag < 0 then
begin // trying to cause an exception if it is not a
process number label
                outputPanel.Controls[i].free;
            end;
        except
            end;
        end;
    end;
    outputPanel.Repaint;

    for i := 0 to high(BestTreeMacs) do begin
        Shape := TShape.Create(outputPanel);
        with Shape do begin
            Visible := False;

```

```

        Height := 20;
        Width :=
Round(BestTreeMacs[i][high(BestTreeMacs[i])] * 12.5);
        Top := 50 + i * (Height + 5);
        Left := 50;
        Brush.Color := clGray;
        Parent := OutputPanel;
        Visible := True;
    end;
    for j := 1 to high(BestTreeMacs[i]) do begin
        if (j - 1) mod 3 = 1 then begin
            StartTime := BestTreeMacs[i][j - 1];
            EndTime := BestTreeMacs[i][j];
            // draw the box representing these start and
end times for the machine i
            Shape := TShape.Create(OutputPanel);
            with Shape do begin
                Visible := False;
                Height := 20;
                Width := Round((EndTime - StartTime) *
12.5);
                Top := 50 + i * (Height + 5);
                Left := 50 + Round(StartTime * 12.5);
                Brush.Color := clRed;
                Parent := OutputPanel;
                Visible := True;
            end;
            ProcessNum := TLabel.Create(OutputPanel);
            with ProcessNum do begin
                Visible := False;
                Caption := FormatFloat('0',
BestTreeMacs[i][j - 2]);
                Font.Color := clBlack;
                Color := clRed;
                Left := 50 + Round(StartTime * 12.5) +
(Round((EndTime - StartTime) * 12.5) div 2) - Width
div 2;
                Top := 50 + i * (Shape.Height + 5) +
(Shape.Height - Height) div 2;
                Tag := -1;
                Parent := OutputPanel;
                Visible := True;
            end;
        end;
    end;
end;
end;
end;

```

```

end;

procedure TShopForm.FormResize(Sender: TObject);
begin
  ItemsBox.Height := (3 * Height) div 8;
  OutputPanel.Height := (3 * Height) div 8;
  // ActionsBox.Top := ClientHeight -
  ActionsBox.Height;
  // SetupBox.Top := ClientHeight - SetupBox.Height;
end;

procedure TShopForm.MutateClick(Sender: TObject);
var
  i, j, k, r, ass           : Integer;
  WorstScore, BestScore   : Single;
  BestIndex, WorstIndex   : Integer;
  Weights                  : Array of Single;
  lnSumOfScores            : Double;
  SumNumerators            : Double;
  numOfKids                : Array of Integer;
  totalKids                : Integer;
  Nodes                    : Array[0..1] of
IXMLDOMEElement;
  Parents                  : Array[0..1] of
IXMLDOMEElement;
  dirs                     : Array[0..1] of string;
  NodeNumbers              : Array[0..1] of Integer;
  NodesClash               : Boolean;
  CurrentTree              : IXMLDOMEElement;
  DonorTree, DonorNode     : IXMLDOMEElement;
  DonorNodeList            : Array of Integer;
  RandomVal                : Single;
  DonorFound               : Boolean;
  BestCloned               : Boolean;
  OneBestCloned            : Boolean;
begin
  // backup old tree
  LastGen :=
IXMLDOMEElement(Solutions.DefaultInterface.documentElement.cloneNode(true));
  if StoreGens.Checked then // there is the option of
not storing all data as it can use hundreds of MB of
memory
AllGens.DefaultInterface.documentElement.appendChild(S

```

```

olutions.DefaultInterface.documentElement.cloneNode(true));
// destroy old tree so new items can be written
FreeAndNil(Solutions);
Solutions := TDOMDocument.Create(Self);
Solutions.DefaultInterface.documentElement :=
Solutions.DefaultInterface.createElement('solutions');

Solutions.DefaultInterface.insertBefore(Solutions.DefaultInterface.createProcessingInstruction('xml',
'version="1.0" encoding="UTF-8"),
Solutions.DefaultInterface.documentElement);
// get top and bottom scores
BestScore := -1;
WorstScore := -1;
for i := 0 to high(BestScoresGen) do begin
    if (BestScoresGen[i] < BestScore) or (BestScore <
0) then begin
        BestScore := BestScoresGen[i];
        BestIndex := i;
    end;
    if (BestScoresGen[i] > WorstScore) or (WorstScore
< 0) then begin
        WorstScore := BestScoresGen[i];
        WorstIndex := i;
    end;
end;
// normalise scores as starting from 0
for i := 0 to high(Scores) do begin
    BestScoresGen[i] := BestScoresGen[i] +
NormValue.Value - BestScore;
end;
WorstScore := WorstScore + NormValue.Value -
BestScore;
BestScore := NormValue.Value;
// get weights: (1 - ln(score[i]) / ln(totalScore) /
sum of numerators
setLength(Weights, Length(BestScoresGen));
lnSumOfScores := 0;
for i := 0 to high(BestScoresGen) do begin
    lnSumOfScores := lnSumOfScores + BestScoresGen[i];
end;
lnSumOfScores := ln(lnSumOfScores);

for i := 0 to high(Scores) do begin // Version 2
weighting only

```

```

    Scores[i] := 1 + WorstScore - Scores[i];
end;

for i := 0 to high(BestScoresGen) do begin
    SumNumerators := SumNumerators + power(0.80 +
ScaleValue.AsInteger * 0.02, BestScoresGen[i]);
// Version 2 weighting
//    SumNumerators := SumNumerators + 1 -
ln(Scores[i]) / lnSumOfScores; // Version 1 weighting
end;
for i := 0 to high(BestScoresGen) do begin
    Weights[i] := power(0.80 + ScaleValue.AsInteger *
0.02, BestScoresGen[i]) / SumNumerators;
// Version 2 weighting
//    Weights[i] := (1 - ln(Scores[i]) /
lnSumOfScores) / SumNumerators; // Version 1
weighting
end;
// determine how many of each tree will be mutated and
passed to the next generation
    setLength(NumOfKids, Length(BestScoresGen));
    totalKids := 0;
    for i := 0 to high(BestScoresGen) do begin
        NumOfKids[i] := Round(Weights[i] *
SampleSize.AsInteger);
        totalKids := totalKids + NumOfKids[i];
    end;
    while totalKids < SampleSize.AsInteger do begin
        Inc(NumOfKids[round(random(High(NumOfKids)))]);
        Inc(totalKids);
    end;
    while totalKids > SampleSize.AsInteger do begin
        if NumOfKids[WorstIndex] > 0 then
            NumOfKids[WorstIndex] := NumOfKids[WorstIndex] -
1
        else begin
            RandomVal := BestIndex;
            while (RandomVal = BestIndex) or
(NumOfKids[Round(RandomVal)] = 0) do
                RandomVal := Random(High(NumOfKids));
            NumOfKids[Round(RandomVal)] :=
NumOfKids[Round(RandomVal)] - 1;
        end;
        totalKids := totalKids - 1;
    end;
// mutate trees: for each tree: begin

```

```

OneBestCloned := false; // one copy of the first
tree that has the best makespan will be passed through
for i := 0 to high(BestScoresGen) do begin
  if random < 0.5 then
    BestCloned := False // 50% chance of cloning a
tree that has the best makespan
  else
    BestCloned := True;
    for j := 0 to NumOfKids[i] - 1 do begin
      CurrentTree :=
IXMLDOMEElement(LastGen.ChildNodes[i].cloneNode(True));
      if (BestScoresGen[i] = BestScore) and (not
(BestCloned and OneBestCloned)) then begin// making
sure that at least one tree that has
        BestCloned := true; // the best score is
passed unchanged to the next generation.
        OneBestCloned := true; // It ensures that the
best score for each generation does not get worse
      end else begin
//      choose node mutation or sub-tree mutation
        RandomVal := random;
        if RandomVal <= NodeMutationRate.Value then
begin
          try // example assertion code
            for ass := 0 to MachinesBox.AsInteger *
JobsBox.AsInteger - 1 do
              assert(getNodeByNumber(currentTree, ass) <>
nil, 'The node with number ' + IntToStr(ass) + ' is
missing from currentTree in the NODE mutation section
of mutategclick.');
```

```

          except
            on E: EAssertionFailed do
              MessageDlg(E.Message, mtWarning, [mbOK],
0);
          end;

//      if node mutation then begin
//      select nodes
        for k := 0 to 1 do begin
          nodeNumbers[k] :=
round(random(MachinesBox.AsInteger * JobsBox.AsInteger
- 1));
          while (k = 1) and (nodenumbers[k] =
nodenumbers[k - 1]) do

```

```

        nodeNumbers[k] :=
round(random(MachinesBox.AsInteger * JobsBox.AsInteger
- 1));
        Nodes[k] :=
getNodeByNumber(CurrentTree, nodeNumbers[k]);
        end;
//      swap values

Nodes[0].attributes.getNamedItem('number').text :=
IntToStr(nodeNumbers[1]);

Nodes[1].attributes.getNamedItem('number').text :=
IntToStr(nodeNumbers[0]);
        end else if RandomVal <=
NodeMutationRate.Value + SubMutationRate.Value then
begin
//      if subtree mutation then begin
//      select nodes
        for k := 0 to 1 do begin
            nodeNumbers[k] :=
round(random(MachinesBox.AsInteger * JobsBox.AsInteger
- 1));
            while getNodeByNumber(currentTree,
nodeNumbers[k]).attributes.getNamedItem('dir').text =
'top' do
                nodeNumbers[k] :=
round(random(MachinesBox.AsInteger * JobsBox.AsInteger
- 1)); // preventing top node from being chosen

            if k = 1 then begin
                NodesClash := True;
                while NodesClash do begin
                    NodesClash := False;
//      Choose a node
                    Nodes[k] :=
getNodeByNumber(CurrentTree, nodeNumbers[k]);
//      if node and Nodes[0] are directly
related then choose again
                    if isRelated(Nodes[0], Nodes[1]) then
begin
                        NodesClash := True;
                        nodeNumbers[k] :=
round(random(MachinesBox.AsInteger * JobsBox.AsInteger
- 1));

```

```

        while getNodeByNumber(currentTree,
nodeNumbers[k]).attributes.getNamedItem('dir').text =
'top' do
            nodeNumbers[k] :=
round(random(MachinesBox.AsInteger * JobsBox.AsInteger
- 1)); // preventing top node from being chosen
        end;
        end;
        end else
            Nodes[k] :=
getNodeByNumber(currentTree, nodeNumbers[k]);
            Parents[k] :=
IXMLDOMElement(Nodes[k].parentNode);
//          save left/right values
            dirs[k] :=
Nodes[k].attributes.getNamedItem('dir').text;

        end;
//          prune nodes
        for k := 0 to 1 do
            Parents[k].removeChild(Nodes[k]);

//          swap positions
        for k := 0 to 1 do begin
//          fix dir values
            if dirs[k] = 'left' then begin
                if Parents[k].childNodes.length = 0 then
begin
                    Nodes[abs(k -
1)].attributes.getNamedItem('dir').text := dirs[k];
                    Parents[k].appendChild(Nodes[abs(k -
1)].cloneNode(true));
                end else begin
                    Nodes[abs(k -
1)].attributes.getNamedItem('dir').text := dirs[k];
                    Parents[k].insertBefore(Nodes[abs(k -
1)].cloneNode(true), Parents[k].firstChild);
                end;
            end else begin
                Nodes[abs(k -
1)].attributes.getNamedItem('dir').text := dirs[k];
                Parents[k].appendChild(Nodes[abs(k -
1)].cloneNode(true));
            end;
        end;
    end;
end;

```

```

        end;
        end else if RandomVal <=
NodeMutationRate.Value + SubMutationRate.Value +
GraftRate.Value then begin
        DonorFound := false;
        while not DonorFound do begin
//          get donor tree
            r := i;
            while r = i do
                r := Round(random(SampleSize.AsInteger -
1));
                DonorTree :=
IXMLDOMEElement(LastGen.childNodes[r].cloneNode(true));
//          extract donor branch -> compile numbers
into list
                r := round(random(MachinesBox.AsInteger *
JobsBox.AsInteger - 1));
                while getNodeByNumber(donorTree,
r).attributes.getNamedItem('dir').text = 'top' do
                    r := round(random(MachinesBox.AsInteger
* JobsBox.AsInteger - 1)); // preventing top node
from being chosen
                    DonorNode := getNodeByNumber(DonorTree,
r);
                    TableInc := 0;
                    setLength(DonorNodeList,
MachinesBox.AsInteger * JobsBox.AsInteger);
                    for r := 0 to high(DonorNodeList) do
                        DonorNodeList[r] := -1;
                    getItems(DonorNode, DonorNodeList, 0);
                    for r := 0 to high(DonorNodeList) do
                        if DonorNodeList[r] = -1 then break;
                    setLength(DonorNodeList, r);
//          check that all nodes in donor branch can be
safely removed from the recipient
                    DonorFound := true;

                    for r := 0 to High(DonorNodeList) do begin
                        if not CanNodeBeDeleted(CurrentTree,
DonorNodeList[r]) then begin
                            DonorFound := false;
                            break;
                        end;
                    end;
                end;
            end;
        end;
    end;
end;
end;

```

```

//      traverse recipient tree, removing each node
//      that appears in the donor branch list and promoting
//      random children to fill the parent node's role
      try
        CurrentTree := deleteItems(CurrentTree,
DonorNodeList);
      except
        on E: Exception do
          MessageDlg(E.Message, mtWarning, [mbOK],
0);
        end;

//      graft donor tree at random point of recipient
      graft(CurrentTree, donorNode);
    end;
  end;
//  assign new tree to new sample

Solutions.DefaultInterface.documentElement.appendChild
(currentTree);
  end;
end;

  GenNum := GenNum + 1;
end;

function TShopForm.getNode(Element : IXMLDOMEElement;
var number : Integer) : IXMLDOMEElement;
var
  i : Integer;
begin
  Result := nil;
  if number = 0 then begin
    Result := Element;
    exit;
  end else begin
    for i := 0 to Element.childNodes.length - 1 do
begin
  if Result = nil then begin
    number := number - 1;
    Result :=
getNode(IXMLDOMEElement(Element.Children[i]),
number);
  end else
    exit;
end;
end;
end;

```

```

    end;
end;

function TShopForm.canNodeBeDeleted(Element :
IXMLDOMEElement; number : Integer) : boolean;
var
    // checks to see if node with number
attribute = number is a leaf or parent of a leaf
    i, j : Integer;
begin
    Result := false;

    if (Element.attributes.getNamedItem('number').text =
intToStr(number)) and (Element.childNodes.length = 0)
then begin
        Result := true;
        exit;
    end;
    if Element.parentNode <> nil then begin
        if
(Element.parentNode.attributes.getNamedItem('number').
text = intToStr(number)) and
(Element.childNodes.length < 2) then begin
            Result := true;
            exit;
        end else if
Element.parentNode.attributes.getNamedItem('number').t
ext = intToStr(number) then
            exit;
        end;

        for i := 0 to Element.childNodes.length - 1 do begin
            Result :=
canNodeBeDeleted(IXMLDOMEElement(Element.childNodes[i])
, number);
            if Result then exit;
        end;
    end;

function TShopForm.isRelated(x, y : IXMLDOMEElement) :
Boolean;
begin
    Result := isAncestor(x, y) or isAncestor(y, x);
end;

function TShopForm.isAncestor(x, y : IXMLDOMEElement) :
Boolean;

```

```

begin
  if (not (x.attributes.getNamedItem('dir').text =
'top')) and (y.attributes.getNamedItem('dir').text =
'top') then
    Result := False
  else if (x.attributes.getNamedItem('number').text) =
(y.attributes.getNamedItem('number').text) then
    Result := True
  else
    Result := isAncestor(x,
IXMLDOMElement(y.parentNode));
end;

procedure TShopForm.MutationRateChange(Sender:
TObject);
var
  relativeSizes : array[0..1] of single;
begin
//  if SubMutationRate.Value + NodeMutationRate.Value
+ GraftRate.Value > 1 then begin
  if (Sender = SubMutationRate) or (Sender =
CloneRate) then begin
    RelativeSizes[0]           :=
NodeMutationRate.Value / (GraftRate.Value +
NodeMutationRate.Value);
    RelativeSizes[1]           := GraftRate.Value /
(GraftRate.Value + NodeMutationRate.Value);
    NodeMutationRate.OnChange := nil;
    GraftRate.OnChange        := nil;
    NodeMutationRate.Value    := ((1 -
CloneRate.Value) - SubMutationRate.Value) *
RelativeSizes[0];
    GraftRate.Value           := ((1 -
CloneRate.Value) - SubMutationRate.Value) *
RelativeSizes[1];
    NodeMutationRate.OnChange := MutationRateChange;
    GraftRate.OnChange        := MutationRateChange;
  end else if Sender = NodeMutationRate then begin
    RelativeSizes[0]           :=
SubMutationRate.Value / (GraftRate.Value +
SubMutationRate.Value);
    RelativeSizes[1]           := GraftRate.Value /
(GraftRate.Value + SubMutationRate.Value);
    SubMutationRate.OnChange  := nil;
    GraftRate.OnChange        := nil;
  end;
end;

```

```

        SubMutationRate.Value      := ((1 -
CloneRate.Value) - NodeMutationRate.Value) *
RelativeSizes[0];
        GraftRate.Value            := ((1 -
CloneRate.Value) - NodeMutationRate.Value) *
RelativeSizes[1];
        SubMutationRate.OnChange := MutationRateChange;
        GraftRate.OnChange       := MutationRateChange;
    end else begin
        RelativeSizes[0]          :=
NodeMutationRate.Value / (SubMutationRate.Value +
NodeMutationRate.Value);
        RelativeSizes[1]          :=
SubMutationRate.Value / (SubMutationRate.Value +
NodeMutationRate.Value);
        NodeMutationRate.OnChange := nil;
        SubMutationRate.OnChange  := nil;
        NodeMutationRate.Value    := ((1 -
CloneRate.Value) - GraftRate.Value) *
RelativeSizes[0];
        SubMutationRate.Value     := ((1 -
CloneRate.Value) - GraftRate.Value) *
RelativeSizes[1];
        NodeMutationRate.OnChange := MutationRateChange;
        SubMutationRate.OnChange  := MutationRateChange;
    end;
// end;
end;

function TShopForm.deleteItems(root : IXMLDOMEElement;
var donorNodeList : array of Integer) :
IXMLDOMEElement;
var
    i, j          : Integer;
    parent        : IXMLDOMEElement;
    children      : array of IXMLDOMEElement;
    childToPromote : Integer;
    rootdir       : string;
    ChildrenLength : Integer;
    BufferElement  : IXMLDOMEElement;
begin
    Result := IXMLDOMEElement(root.cloneNode(false));
    for i := 0 to high(donorNodeList) do begin
        if
StrToInt(Result.attributes.getNamedItem('number').text
) = donorNodeList[i] then begin

```

```

        rootdir :=
Result.attributes.getNamedItem('dir').text;
//  get the current node's children
    setlength(children, root.childNodes.length);
    for j := 0 to high(Children) do
        children[j] :=
IXMLDOMEElement(root.childNodes[j].cloneNode(true));
//  disconnect children from root node
    while root.hasChildNodes do
        root.removeChild(root.firstChild);
//  if there is at least one child, choose a child to
be promoted
    ChildrenLength := Length(Children);
    if childrenlength = 1 then
        childToPromote := 0
    else if childrenlength > 1 then begin
//  look at each child and promote the first child
that has at most one child of its own
        for j := 0 to childrenlength - 1 do begin
            if children[j].childNodes.length <= 1 then
begin
                childToPromote := j;
                break;
            end;
        end;
    end else
        childToPromote := -1;
//  assign the child to the root level in either the
top, left or right position as appropriate
    if childToPromote >= 0 then begin

Children[ChildToPromote].attributes.getNamedItem('dir'
).text := rootdir;
        Result :=
IXMLDOMEElement(Children[ChildToPromote].cloneNode(true
));
    end else
        Result := nil;

//  if there is still one child left, add it to
either the left or right position as appropriate
    if Childrenlength = 2 then begin
        if Result.childNodes.length = 1 then begin //
if the node that was promoted already has one child

```

```

        if
Result.firstChild.attributes.getNamedItem('dir').text
= 'left' then begin // and if the child is on the left

Result.appendChild(Children[abs(ChildToPromote -
1)].cloneNode(true)); // add the original root node's

Result.lastChild.attributes.getNamedItem('dir').text
:= 'right';          // other child to the right
        end else begin                                //
and if the child is on the right

Result.insertBefore(Children[abs(ChildToPromote -
1)].cloneNode(true), Result.firstChild); // add the
original root node's

Result.firstChild.attributes.getNamedItem('dir').text
:= 'left';          // other child to
the left
        end;
        end else // or if the node that was promoted
had no children, then add the original root node's
other child

Result.appendChild(Children[abs(ChildToPromote -
1)].cloneNode(true)); // to whichever side it was
already on
        end;
// if the above code is executed, then call
deleteItems(newRoot, table); exit; to avoid the code
below

        if ChildToPromote >= 0 then
            Result :=
deleteItems (IXMLDOMEElement (Result.cloneNode (true)),
donorNodeList);
            exit;
        end;
    end;

    ChildrenLength := root.childNodes.length;
    i := 0;
    while i < childrenLength do begin
        BufferElement :=
deleteItems (IXMLDOMEElement (root.childNodes[i].cloneNod
e(true)), donorNodeList);

```

```

    if BufferElement <> nil then
        Result.appendChild(BufferElement);
        i := i + 1;
    end;
end;

function TShopForm.graft(tree, branch :
IXMLDOMEElement) : boolean;
var
    // grafting takes place at the first
    // available position, thereby ensuring that trees take
    // less space
    i : Integer; // than if they were allowed to branch
    // out in any way
begin
    Result := true;
    if tree.childNodes.length = 0 then
        tree.appendChild(branch.cloneNode(true))
    else if tree.childnodes.length = 1 then begin
        if
tree.firstChild.attributes.getNamedItem('dir').text =
'left' then begin
            tree.appendChild(branch.cloneNode(true));

tree.lastChild.attributes.getNamedItem('dir').text :=
'right';
            end else begin
                tree.insertBefore(branch.cloneNode(true),
tree.firstChild);

tree.firstChild.attributes.getNamedItem('dir').text :=
'left';
            end;
        end else begin
            Result := false;
            for i := 0 to tree.childnodes.length - 1 do begin
                Result :=
graft(IXMLDOMEElement(tree.childnodes[i]), branch);
                if result then exit;
            end;
        end;
    end;

end;

procedure TShopForm.SaveCSVClick(Sender: TObject);
var
    CSV : TFileStream;

```

```

    csvstring : String;
    i          : Integer;
begin
    // put in data
    here at some point
        csvstring := 'Sample Size,' +
        IntToStr(SampleSize.AsInteger) + ',Time taken (min),'
+ FormatFloat('0.000', ((end_time - start_time) /
ticks_per_second) / 60) + #13#10'Generation,Best
Score,Average Score'#13#10;
        for i := 0 to high(BestScores) do begin
            csvString := csvString + IntToStr(i) + ',' +
FloatToStr(BestScores[i]) + ',' +
FloatToStr(AverageScores[i]) + #13#10;
        end;

    try
        try
            if Filename = '' then
                CSV := TFileStream.Create('.\output.csv',
fmCreate or fmOpenWrite)
            else
                CSV :=
TFileStream.Create(ChangeFileExt(Filename, '.nsg.' +
formatfloat('0.00', NodeMutationRate.Value) + '-' +
formatfloat('0.00', SubMutationRate.Value) + '-' +
formatfloat('0.00', GraftRate.Value) + ' -- ' +
IntToStr(runNumber) + '.csv' ), fmCreate or
fmOpenWrite);
            except
                CSV := TFileStream.Create('c:\output.csv',
fmCreate or fmOpenWrite);
            end;
            CSV.Write(PChar(csvstring)^,Length(csvstring));
        except
            MessageDlg('Could not create the csv
file.'#13#10'Please check that there is enough disk
space and that the disk is not write-protected.',
mtError, [mbOK], 0);
        end;
        FreeAndNil(CSV);
    end;

procedure TShopForm.DoXGensClick(Sender: TObject);
var
    i : Integer;
begin

```

```

try
  QueryPerformanceCounter(start_time);
  for i := 0 to GensToDo.AsInteger - 1 do begin
    if GenNum = 0 then
      MakeGen0ButtonClick(Self)
    else begin
      try
        MutateClick(Self);
      except
        MessageDlg('Error in MutateClick.', mtError,
[mbOK], 0);
      end;
    end;
    try
      ScoreSolutionsClick(Self);
    except
      MessageDlg('Error in ScoreSolutionsClick.',
mtError, [mbOK], 0);
    end;
  end;
  if (Sender <> TestFileButton) then
    DrawBestClick(Self);
  QueryPerformanceCounter(end_time);
  SaveCSVClick(Self);
  try
    if (Sender = DoXGens) and StoreGens.Checked then

AllGens.DefaultInterface.save('.\allGens.xml');
  except
    try
      if StoreGens.Checked then

AllGens.DefaultInterface.save('c:\allGens.xml');
    except
      MessageDlg('Could not create
allGens.xml.'#13#10'Please check that there is enough
disk space and that the disk is not write-protected.',
mtError, [mbOK], 0);
    end;
  end;
  except
  end;
end;

procedure TShopForm.GensToDoChange(Sender: TObject);
begin

```

```

    DoXGens.Caption := 'Do ' +
IntToStr(GensToDo.AsInteger) + ' Gens';
end;

procedure TShopForm.LoadButtonClick(Sender: TObject);
var
    InFile      : TFileStream;
    InString    : TStringStream;
    ch          : char;
    curNum      : String;
    readingNum  : Boolean;
    numsRead   : Integer;
    s           : String;
begin
    openFileDialog.Execute;
    if openFileDialog.FileName = '' then exit;
    FileName := openFileDialog.FileName;
    try
        numsRead := 0;
        readingNum := False;
        InFile :=
TFileStream.Create(openDialog.FileName, fmOpenRead);
        InString := TStringStream.Create(s);
        InString.CopyFrom(Infile, 0);
        InFile.Free;
        InString.Position := 0;
        while not (InString.Position = Instring.Size) do
begin
    InString.Read(ch, 1);
    try
        if StrToInt(string(ch)) > -1000 then begin
            if not readingNum then begin
                readingNum := true;
                curNum := '';
            end;
            curNum := curNum + ch;
        end;
    except
        if readingNum then begin
            readingNum := false;
            if numsRead = 0 then
                JobsBox.Value := StrToInt(curNum)
            else if numsRead = 1 then begin
                MachinesBox.Value := StrToInt(curNum);
                ChangeItemsClick(Self);
            end else begin

```

```

        Items[((numsRead div 2) - 1) mod
MachinesBox.AsInteger][((numsRead div 2) - 1) div
MachinesBox.AsInteger][(numsRead + 1) mod 2].Value :=
StrToInt(curNum);
        end;
        numsRead := numsRead + 1;
    end;
end;
end;
finally
//    Infile.Free;
    InString.Free;
end;

end;
procedure TShopForm.TestFileButtonClick(Sender:
TObject);
begin
    SubMutationRate.MaxValue := 2;
    NodeMutationRate.MaxValue := 2;
    GraftRate.MaxValue := 2;
//    NodeMutationRate.Value := 0.0;
    while NodeMutationRate.Value <= 0.9 do begin
        SubMutationRate.Value := 0;
        while SubMutationRate.Value <= 1 -
NodeMutationRate.Value do begin
            GraftRate.Value := 1 - (NodeMutationRate.Value +
SubMutationRate.Value);
            DoXGensClick(TestFileButton);
            FormCreate(ChangeItems);
            SubMutationRate.Value := SubMutationRate.Value +
0.1;
        end;

        NodeMutationRate.Value := NodeMutationRate.Value +
0.05;
    end;
    SubMutationRate.MaxValue := 1;
    NodeMutationRate.MaxValue := 1;
    GraftRate.MaxValue := 1;
end;

procedure TShopForm.runtestsbuttonClick(Sender:
TObject);
var
    i, j : Integer;

```

```
begin
  for j := 0 to 4 do begin
    runNumber := j;
    GenNum := 0;
    for i := 0 to 4 do begin
      ScaleValue.Value := 5 + i;
      NormValue.Value := 1 + 7 * i;
      DoXGensClick(TestFileButton)
    end;
  end;

end;

initialization
  // get ticks/sec

QueryPerformanceFrequency(TLargeInteger(ticks_per_second));
end.
```