USING DISTRIBUTED COMPUTING TO IMPROVE THE PERFORMANCE OF GENETIC
ALGORITHMS FOR JOB SHOP SCHEDULING PROBLEMS

A thesis presented to
the faculty of
Fritz J. and Dolores H. Russ College of Engineering and Technology
of
Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Nihar Shah
August 2004
This thesis entitled

USING DISTRIBUTED COMPUTING TO IMPROVE THE PERFORMANCE OF GENETIC
ALGORITHMS FOR JOB SHOP SCHEDULING PROBLEMS

By
Nihar Shah

has been approved for
the Department Of Industrial And Manufacturing Systems Engineering and
the Fritz J. and Dolores H. Russ College of Engineering and Technology by

David A Koonce

Associate Professor Of Industrial And Systems Engineering

Dennis Irwin

Dean, Fritz J. and Dolores H. Russ College of Engineering and Technology
Scheduling problems arise in diverse areas such as flexible manufacturing systems, logistics, and network systems and so on. A job shop scheduling problem (JSP) is among the toughest scheduling problems due to the huge number of possible solutions for every problem. There is no efficient conventional optimization algorithm that can guarantee an optimal solution in polynomial time. Due to this inherent intractability, heuristic procedures such as genetic algorithms offer an attractive way of solving these problems. Even with genetic algorithms, larger job shop problems lead to highly time consuming computations due to the vast solution space. This research tries to use the principles of distributed computing to improve solution time in case of such large problems. This document proposes software developed in Java programming language to solve job shop scheduling problems using genetic algorithms by simultaneously utilizing the processing capabilities of several networked computers. The software is based on the client-server model, where the server distributes the computationally intensive task of crossover, mutation and makespan calculation for chromosomes to remote clients. Testing has been done to determine whether this approach is useful in reducing computation time in case of different sizes of job shop scheduling problems.
and different types of hardware configuration. Results of the testing are discussed at the end of the document.

Approved:

David A Koonce
Associate Professor
# Table of Contents

Abstract .................................................................................................................................................. 3  

1. Introduction ...................................................................................................................................... 13  
   1. 1. The Problem ............................................................................................................................ 13  
   1. 2. Research Objective .................................................................................................................. 14  
   1. 3. Thesis Structure ....................................................................................................................... 15  

2. Background ........................................................................................................................................ 16  
   2. 1. The job shop problem .............................................................................................................. 16  
      2. 1. 1. Example of a job shop problem ....................................................................................... 16  
      2. 1. 2. Solving the job shop problem ....................................................................................... 17  
   2. 2. Genetic Algorithms .................................................................................................................. 19  
      2. 2. 1. What is a genetic algorithm? ............................................................................................. 19  
      2. 2. 2. The genetic algorithm mechanism .................................................................................. 19  
      2. 2. 3. Genetic operators ............................................................................................................. 22  
      2. 2. 4. Terminating the genetic algorithm .................................................................................. 24  
      2. 2. 5. Genetic algorithms in scheduling .................................................................................... 24  
      2. 2. 6. Genetic algorithms in job shop scheduling ....................................................................... 25  
   2. 3. Parallelization ............................................................................................................................ 25  
      2. 3. 1. Single Instruction Multiple Data (SIMD) ........................................................................... 26  
      2. 3. 2. Vector Processing ............................................................................................................ 27  
      2. 3. 3. Multiple Instruction Multiple Data (MIMD) .................................................................... 28  
      2. 3. 4. Threads ............................................................................................................................. 29
2. 4. Distributed computing ..................................................................................30

2. 4. 1. Distributed algorithms ...........................................................................30

2. 4. 2. Distributed GA implementations .............................................................32

3. Distributed Genetic Algorithm Software ..........................................................34

3. 1. The distributed genetic algorithm model .........................................................34

3. 2. Genetic algorithm features ...........................................................................38

3. 2. 1. Chromosome representation ..................................................................38

3. 2. 2. Genetic Operators ...............................................................................39

3. 2. 3. Evaluation function .............................................................................42

3. 2. 4. Selection strategies .............................................................................43

4. Testing and Results .............................................................................................46

4. 1. Approach ....................................................................................................46

4. 2. Case 1 ........................................................................................................47

4. 2. 1. Test Configuration ...............................................................................47

4. 2. 2. Results ................................................................................................48

4. 3. Case 2 ........................................................................................................49

4. 3. 1. Test Configuration ...............................................................................49

4. 3. 2. Results ................................................................................................50

4. 4. Case 3 ........................................................................................................53

4. 4. 1. Test Configuration ...............................................................................53

4. 4. 2. Results ................................................................................................54

4. 5. Case 4 ........................................................................................................55

4. 5. 1. Test Configuration ...............................................................................55
Class: edu.ohiou.ga.jobshop.JSPGeneration......................................................... 114
Class: edu.ohiou.ga.jobshop.JSPHandler ............................................................. 116
Class: edu.ohiou.ga.jobshop.JSPJob................................................................. 120
Class: edu.ohiou.ga.jobshop.JSPMachine......................................................... 122
Class: edu.ohiou.ga.jobshop.JSPOperation ..................................................... 123
Class: edu.ohiou.ga.jobshop.JSPOrderCrossover ............................................. 125
Class: edu.ohiou.ga.jobshop.JSPOrderMutation.............................................. 128
Class: edu.ohiou.ga.jobshop.JSPScenario ...................................................... 130
Class: edu.ohiou.ga.server.ServerControl ...................................................... 135
Class: edu.ohiou.ga.server.ServerEngine......................................................... 137
Class: edu.ohiou.ga.GaException .................................................................... 147
Class: edu.ohiou.ga.GaParameters................................................................. 148
Class: edu.ohiou.ga.GaResults.................................................................... 150
Class: edu.ohiou.ga.ResultData .................................................................... 152
List of Figures

Figure 1: A Gantt chart representation of a job shop schedule. ..................................17
Figure 2: A general procedure for genetic algorithms..................................................20
Figure 3: The general structure of a genetic algorithm.................................................21
Figure 4: Applying crossover operation between two chromosomes............................22
Figure 5: Applying the mutation operator on a chromosome. .....................................23
Figure 6: Parallelization of a population in a genetic algorithm. .................................35
Figure 7: The client-server architecture of the genetic algorithm. .................................36
Figure 8: Flow-chart describing the operation of the client and the server.....................37
Figure 9: Cutting point probabilities in a chromosome. .............................................39
Figure 10: Transferring genes from the first parent to the offspring............................40
Figure 11: Removal of genes from C₂ to maintain validity of O₁. .................................41
Figure 12: Copying genes from C₂ to O₁ to produce final offspring.............................41
Figure 13: Swapping two non-identical genes in a chromosome. .................................42
Figure 14: Makespan calculation of a chromosome using FIFO builder. .......................43
Figure 15: Summary of test results obtained in Case 1. .............................................58
Figure 16: Summary of test results obtained in Case 2. .............................................59
Figure 17: Summary of test results obtained in Case 3. .............................................59
Figure 18: Summary of timed test results obtained in Case 4. .....................................61
Figure 19: Makespan values for different number of clients......................................62
Figure 20: UML Diagram of the edu.ohiou.ga.interfaces package............................71
Figure 21: UML Diagram of the edu.ohiou.ga.server package.................................73
Figure 22: UML diagram for the *edu.ohiou.ga.jobshop* package.................................75
Figure 23: DTD for the input XML file that stores the problem data. .........................76
Figure 24: XML file for the ft06 (6 x 6) problem. .....................................................77
Figure 25: UML structure of the *edu.ohiou.ga.client* package. ................................79
Figure 26: The genetic algorithm server in operation..................................................81
Figure 27: The genetic algorithm client in operation. ...............................................81
Figure 28: The genetic algorithm server at the end of the GA cycle. ...........................82
# List of Tables

Table 1: Process plan for a 3 x 3 job shop problem. ..................................................17
Table 2: The Fisher and Thompson 6x6 instance (ft06). ............................................38
Table 3: Chromosome representation of the solution to the ft06 problem. .................38
Table 4: Chromosomes in a sample population. ........................................................44
Table 5: Calculation of reproduction probabilities of chromosomes. ............................44
Table 6: Parameters used in testing the genetic algorithm. ..........................................47
Table 7: Hardware and software configuration for Case 1. .........................................47
Table 8: Time required for solving the 6x6 problem...................................................48
Table 9: Time required for solving the 10x10 problem...............................................49
Table 10: Time required for solving the 20x10 problem. ............................................49
Table 11: Hardware and software configuration for Case 2. .......................................50
Table 12: Time required for solving the 6x6 problem...................................................51
Table 13: Time required for solving the 10x10 problem...............................................51
Table 14: Time required for solving the 20x10 problem. ............................................51
Table 15: Time required for solving the 30x10 problem. ............................................52
Table 16: Time required for solving the 20x20 problem. ............................................52
Table 17: Time required for solving the 50x10 problem. ............................................52
Table 18: Hardware and software configuration of server for Case 3. .........................53
Table 19: Hardware and software configuration of clients for Case 3. .........................53
Table 20: Time required for solving the 30x10 problem. ............................................54
Table 21: Time required for solving the 20x20 problem. ............................................54
Table 22: Time required for solving the 50x10 problem. ............................................55

Table 23 : Time required for solving the 10x10 problem. ...........................................56

Table 24: Makespan for the 10x10 problem for different number of clients. ...............57

Table 25 : One-way ANOVA results for makespan data. ........................................63
1. INTRODUCTION

1. 1. THE PROBLEM

The classical job shop scheduling problem is one of the most complicated scheduling problems with no efficient solution algorithm that can guarantee an optimal solution. The task is to find a schedule of minimum time length that satisfies all the constraints of a given job shop problem. The real difficulty in finding good solutions to these problems is the number of possible solutions for a given scenario. Consider a simple 6 job, 1 machine problem – there are $6! = 720$ possible solutions. If this problem were to have 2 machines, the number of possible solutions would be $6! \times 6! = 518400$. Now consider the 6 job, 6 machine problem and the number of possible schedules would be $(6!)^6$ or over $1.3 \times 10^{17}$ possible solutions. It is not difficult to imagine how tough it would be to solve larger problems such as those involving 10 jobs, 10 machines (approximately $3.95 \times 10^{65}$ possible solutions) and more.

Finding good solutions to these problems has been an area of research for years. Recent heuristic and stochastic search techniques have eased the problem a little, but they are not guaranteed to find good solutions every time. Towards this end, genetic algorithms have been widely used in solving such large, complicated problems because they offer the advantage of exploring a very wide area of the solution space. But, when it comes to larger problems, even these algorithms take time to explore the large number of possible solutions for a given problem.
Researchers have always been looking for answers on how to efficiently solve large problems. This is where the principle of distributed computing comes into the picture. Distributed computing involves breaking up a large problem into several small and simple problems and solving each of these simultaneously on different processors. Distributed computing has been widely used in solving complicated optimization problems in the past, and its usage is increasing rapidly with the advent of cheaper computers being available easily.

1. 2. Research Objective

Genetic algorithms have been used in solving job shop problems in several instances. The only limitation to such cases has been the time it takes to explore the vast solution space for a typical job shop scenario.

This research proposes a distributed client-server genetic algorithm that utilizes the processing power of networked computers and the effectiveness of genetic algorithms in solving these massive problems. The algorithm has been implemented in Java using Java Remote Method Invocation (RMI) to connect network computers and distribute the computationally intensive task of evaluating individual chromosomes to several clients while the server maintains a single population and selects best solutions for recombination and further evolution.
1.3. Thesis Structure

The current chapter introduced the problem that this thesis is trying to solve. Genetic algorithms and job shop problems will be discussed along with solution procedures and past and current research in Chapter 2. Chapter 3 will discuss the proposed client-server architecture of the distributed genetic algorithm. Chapter 4 will be devoted towards testing and results obtained during testing of the proposed genetic algorithm. Conclusions from the test results will be discussed and ideas for future research will be proposed in the Chapter 5.
2. BACKGROUND

2.1. THE JOB SHOP PROBLEM

The definition of a job shop problem as given in Gen and Cheng [1] is: There are \( m \) different machines and \( n \) different jobs to be scheduled. Each job is composed of a set of operations and the operation order on machines is pre-specified. Each operation is characterized by the required machine and the fixed processing time.

Some of the constraints involved in a job shop problem can be given as:

- A job does not visit the same machine twice.
- Jobs do not have precedence constraints among themselves.
- An operation, once started, must be finished, before another operation is started on the same machine.
- Any given machine can process only one job at a time.
- There are no fixed release times or due dates.

2.1.1. EXAMPLE OF A JOB SHOP PROBLEM

A simple 3 x 3 problem is given in Table 1. The data in the table is given in the format (machine, time) and provides the process plan for the job shop.
Table 1: Process plan for a 3 x 3 job shop problem.


<table>
<thead>
<tr>
<th>Job</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1, 3</td>
</tr>
<tr>
<td>2</td>
<td>1, 1</td>
</tr>
<tr>
<td>3</td>
<td>2, 3</td>
</tr>
</tbody>
</table>

Based on the information given in the process plan, several solutions exist for this job shop problem. Each of these solutions gives us a different makespan which is the completion time of the last job in the sequence. The makespan measures the performance of the schedule and is a very effective indicator of machine utilization. The Gantt chart for a typical solution for the problem is given in Figure 1.

Figure 1: A Gantt chart representation of a job shop schedule.

2. 1. 2. SOLVING THE JOB SHOP PROBLEM

Most real world scheduling problems are NP-hard problems, where the solution time / complexity increase exponentially with increase in problem size. Procedures such as brute force evaluation, dynamic programming and integer programming are guaranteed
to find the optimal solution but the time required to solve the problem makes the solution highly impractical.

For this reason, many researchers focus on using heuristic procedures or stochastic search to solve these problems. Job shop problems are most commonly evaluated using dispatching rules such as shortest processing time (SPT), first come first serve (FCFS) or first in first out (FIFO), longest processing time (LPT) and most work remaining (MWKR). For job shop scheduling, none of these procedures can guarantee the optimal makespan but they involve simple and fast calculations and hence, are widely used. These rules along with several other procedures are discussed at length in Panwalkar and Iskander [2]. The shifting bottleneck heuristic developed by Adams et. al. [3] has been the most effective among all the heuristic procedures for solving job shop scheduling problems. Even though heuristic methods are generally effective in scheduling job shops, these techniques tend to get trapped in local search space. This is where stochastic search techniques such as genetic algorithms, simulated annealing, and tabu search provide good schedules by exploring the global search space. The superiority of these algorithms over heuristic procedures has been adequately discussed in Jones and Rabelo [4].

In [5], Ponnambalam et. al. have discussed a tabu search algorithm for determining good schedules for job shop problems. Using makespan as a performance measure, the algorithm generates neighborhoods using adjacent pair wise interchange. In their results, the researchers have found the performance of their proposed algorithm to be
comparable to genetic algorithms and simulated annealing. Steinhofel et. al [6] have proposed a parallel heuristic that employs simulated annealing to solve the classical job shop problem. The problem is represented using the disjunctive graph model and the solution involves minimizing the length of the longest paths.

2. 2. GENETIC ALGORITHMS

2. 2. 1. WHAT IS A GENETIC ALGORITHM?

The simulation of the natural evolutionary process of nature forms the basis of genetic algorithms, which can outperform conventional optimization techniques in certain real world problems. Goldberg [7] defines them as - genetic algorithms are stochastic search techniques based on the mechanism of natural selection and natural genetics. They are very useful in solving problems that cannot be solved easily by conventional optimization techniques.

2. 2. 2. THE GENETIC ALGORITHM MECHANISM

Figure 2 shows the generalized procedure for a typical genetic algorithm and Figure 3 shows the general structure of a genetic algorithm.
begin
  \( t \leftarrow 0; \)
  initialize \( P(t) \);
  evaluate \( P(t) \);
  while (not termination condition) do
    recombine \( P(t) \) to yield \( C(t) \);
    evaluate \( C(t) \);
    select \( P(t+1) \) from \( P(t) \) and \( C(t) \);
    \( t \leftarrow t + 1; \)
  end
end

where
\( t \) – the generation being evaluated
\( P(t) \) – population of GA at generation \( t \)
\( C(t) \) – population formed by genetic operators on \( P(t) \)

Figure 2: A general procedure for genetic algorithms.

Figure 3: The general structure of a genetic algorithm.

(Source: Genetic Algorithms & Engineering Design, Wiley, 1997, Figure 1.1, Page 3)

The algorithm searches for the solution to the problem by performing genetic operations on a population of candidate solutions. This population is evaluated and good solutions are selected to mate and form the next generation. Over a period of time, the overall quality of the population increases and provides an optimal or a near optimal solution.
2. 2. 3. GENETIC OPERATORS

The genetic algorithm explores the solution space by means of two genetic operators – crossover and mutation.

2. 2. 3. 1. Crossover

The candidate solutions in a population recombine using a crossover operation. Crossover is an exploratory mechanism which takes two solutions selected from a population to form two new off-springs by random exchange of substrings between the parents. This operation results in exploration of the global solution space. Consider two chromosomes shown in Figure 4 below. Each chromosome is a potential solution to the 3x3 job shop problem discussed in Table 1.

Figure 4: Applying crossover operation between two chromosomes.
The crossover point is selected randomly and each chromosome is divided into two parts. Part 1 of chromosome 1 combines with part 2 of chromosome 2 to form offspring 1, and part 1 of chromosome 2 combines with part 2 of chromosome 1 to form offspring 2. In certain types of crossover, two parent chromosomes may recombine to form only one offspring, while the remaining genes are discarded. Some crossover operators operate by using multiple cutting points on each chromosome. If the resultant offspring is not a valid solution, the chromosome has to be repaired or made feasible by modifying it.

2. 2. 3. 2. **Mutation**

Mutation is the secondary search operator that explores the local solution space by introducing a small variation in an individual solution. Consider the chromosome shown in Figure 5. The offspring is formed by swapping genes 2 and 8. This is the simplest type of mutation and is commonly used in most genetic algorithms.

![Swapping Genes](image)

**Figure 5: Applying the mutation operator on a chromosome.**
Most genetic algorithms have a very high crossover probability and a very low mutation probability.

2. 2. 4. TERMINATING THE GENETIC ALGORITHM

The genetic algorithm can be terminated in several ways. The most common method is to stop after a predetermined number of generations have been evaluated. Another commonly used method is to terminate when the solution crosses a desired threshold value. Some algorithms terminate when the quality of the population does not improve after a specified number of generations. Another alternative is to halt the algorithm when all the individuals in a generation are identical.

2. 2. 5. GENETIC ALGORITHMS IN SCHEDULING

The applications of genetic algorithms in the field of scheduling are so common that it is beyond the scope of this thesis to review all the available literature in this area. This document attempts to provide a brief insight into some of the research that has already been conducted in this field. Genetic algorithms have been effectively used in machine-part cell-formation by Mak and Wong [8], in cell design with alternative routings by Lee, Luong and Abhary [9] and in designing independent manufacturing cells by Moon and Gen [10]. Previous research by Kannan and Ghosh [11] has demonstrated the method to apply cell scheduling to a job shop to minimize setup times. Order based genetic algorithms have been used in solving NP-hard flow shop problems by Wang, Zhang and

2. 2. 6. Genetic Algorithms in Job Shop Scheduling

Several algorithms have been proposed to solve job shop problems using genetic algorithms. Wang and Zheng [14] have presented a genetic algorithm with a modified mutation operator to enhance the exploring potential of the algorithm. Kumar and Srinivasan [15] have proposed a genetic algorithm and a combination of dispatching rules to solve job shop scheduling problems. Yoshitomi [16] has proposed a method for solving stochastic job shop scheduling problems using genetic algorithms. Koonce and Tsai [17] have used genetic algorithms and data mining to find good sequences for job shop problems.

2. 3. Parallelization

Just as a group of workers can build a house faster than a single worker, many difficult computational problems can be solved in a shorter timeframe through the inclusion of more processors. There is a rich history in computer science of developing parallel solution approaches to many problems. In the simplest case, many processors are used to perform the same operation on multiple data elements; while in the most complex approaches, the solution algorithm in split into independent execution paths, which diverge and merge as necessary.
The determination of the best approach for parallelization depends on the data structure and solution approach for the problem being analyzed. To identify which approach should be taken for this work, an understanding of the approaches available is necessary, as well as the issues involved in converting a serial process into parallel process.

2. 3. 1. SINGLE INSTRUCTION MULTIPLE DATA (SIMD)

While often not a trivial problem, the most basic form of parallelization is to apply a single instruction (or operation) to many data elements at the same time. A typical processor puts a single data element into a register, even if there is room for multiple data elements. A SIMD system packs multiple data elements into a register and performs the same calculation on all of them at the same time.

SIMD architectures contain a single control unit (CU) with multiple processor elements acting as arithmetic units (AU). The arithmetic units are slaves to the control units and cannot fetch or interpret instructions. Their sole purpose is to perform mathematical operations. The control unit interprets instructions and manages task assignment among the available arithmetic units. Although this architecture offers convenience in adding more memory and arithmetic units to the system, the control unit can waste a lot of time managing the memory exchanges between different arithmetic units. SIMD achieves hardware register and data path utilization optimization at the cost of increased
programming complexity. SIMD is an established method for parallelizing problems. It works well in areas where large amounts of data are analyzed. Examples of SIMD utilization include: speech recognition [18], image recognition [19] and image and video encoding [20].

A pipelined SIMD architecture is composed of a pipeline of arithmetic units with shared memory. The pipeline takes different streams of instructions and performs all the operations of an arithmetic unit. The pipeline is a first in first out (FIFO) type of procedure. To take advantage of the pipeline, the data to be evaluated must be stored in different memory modules so the pipeline can be fed with this information as fast as possible. The advantages to this architecture can be found in the speed and efficiency of data processing assuming the above stipulation is met.

2. 3. 2. VECTOR PROCESSING

A vector processor is a processor that can operate on a data set, represented in a vector, in one instruction. The operands to the instructions are complete vectors instead of one element. Vector processors reduce the fetch and decode bandwidth as less numbers of instructions are fetched. These vector instructions operate sequentially on all elements of vector registers. The computation of each result is independent of the computation of previous results, allowing for massive pipelining without compromising on the integrity of data. Supercomputers, such as those from Cray, implement this architecture. Vector processing is widely used in scientific applications such as weather
and satellite imaging, and in multimedia applications such as rendering and video editing.

2. 3. 3. **MULTIPLE INSTRUCTION MULTIPLE DATA (MIMD)**

MIMD architectures have multiple processors. Each processor executes an independent stream (sequence) of machine instructions. The processors execute these instructions by using any accessible data rather than being forced to operate upon a single, shared data stream. Hence, at any given time, an MIMD system can be using as many different instruction streams and data streams as there are processors. MIMD systems may have a shared memory or distributed memory architecture.

There are three types of MIMD systems accessing shared memory. The simplest form is a bus-based system, where all the processors are attached to a common bus which connects to the main memory. Another type is the extended shared memory systems which have a number of independent memory units connected to processors through an interconnection network. Hierarchical MIMD systems have a hierarchy of shared buses to give processors access to each other’s memory.

In distributed memory MIMD machines, each processor has its own individual memory location. The interconnection scheme can be hypercube or mesh network. The hypercube connection network is a system that contains $2^n$ processors with each processor directly connected to $n$ other processors. Here $n$ is the diameter of the
system, which is the minimum number of steps it takes for one processor to send a message to the processor that is the farthest away. In an MIMD distributed memory machine with a mesh interconnection network, processors are placed in a two-dimensional grid and each processor is connected to its four immediate neighbors. Wraparound connections may be provided at the edges of the mesh. One advantage of the mesh interconnection network over the hypercube is that the mesh system need not be configured in powers of two.

2.3.4. THREADS

Another method for parallelizing an application is to identify processing sequences with little or no dependence on other processes. A thread, sometimes referred to as an execution context or a lightweight process, is a single sequential flow of control within a program. Multi-threaded programs allow concurrent flows of control within a single application. Almost all major programming languages have support for threading. When an application spawns multiple threads, the program and operation system schedules the threads based on priorities and availability of processing time. All data, except for the stack and registers, are shared between threads of the same application making them a natural vehicle for expressing tasks that can be broken down into subtasks that can be run cooperatively.
2. 4. DISTRIBUTED COMPUTING

Software based distributed computing is a programming model in which processing occurs in many different nodes in a network. It is an environment where idle processor time and the storage space of networked systems are harnessed to work together on computation-intensive problems. A typical distributed architecture consists of lightweight software agents installed on a number of client systems, with one or more dedicated servers managing these clients. An agent running on the client detects when the system is idle and notifies the server, which in turn, sends out a task to the client. Once the client has completed the task, the results are sent back to the server. The client may run the process in the background or it may be a dedicated client, where the application runs in the foreground. The SETI @ Home [21] project is an ideal example of very large scale distributed computing.

2. 4. 1. DISTRIBUTED ALGORITHMS

Many traditional search algorithms lend themselves to one of the parallelization models. Procedures such as the branch and bound algorithm require exhaustive calculations in case of problems with a huge solution space. Mitra et. al. [22] have proposed a two stage branch and bound algorithm that uses a cluster of Sun workstations. In their two stage approach to the problem, the first stage investigates a number of alternative search trees in the hope of finding a good solution quickly. The best solution obtained at
each branch is broadcast to the other processors involved in the search. In the second stage, they organize parallel search of all the branches of a selected tree.

Topping et. al. [23] have described a parallel processing implementation of neural computing and its application to finite element mesh decomposition. This implementation shows significant improvements in training speed. In this approach, they proposed a master task that creates the neural network model and sends out this information to several worker processors. These worker processors then create their own local neural networks. The root processor keeps all the worker processors in sync and stores the values of all the weights being used in training the network. Using this approach, they were able to demonstrate significant improvement in learning speed.

In another paper, Topping, et. al. [24] have described parallel implementations of neural networks and genetic algorithms in finite element analysis by partitioning the finite element mesh into sub-domains for load balancing.

Niewinski [25] has presented a new approach to overcome the time constraints in using Monte Carlo simulation and has analyzed a high vacuum metrological system using this approach.
2. 4. 2. DISTRIBUTED GA IMPLEMENTATIONS

In his survey of distributed genetic algorithms, Cantu-Paz [26] has classified previous implementations in three broad categories:

- Global single population master-slave algorithms.
- Single population fine grained algorithms.
- Multiple population coarse-grained algorithms.

Most of the previous research has focused on parallel genetic algorithms operating on multiple populations on different machines. Wong, Lee and Leung [27] have proposed a parallel genetic algorithm system named APGAIN which utilizes the spare power of workstations on Internet to solve optimization problems. Sena, Megherbi and Isern [28] have implemented a parallel version of a genetic algorithm and applied it to the well known traveling salesman problem. Alba, Nebro and Troya [29] have outlined issues concerning implementation of parallel genetic algorithms on different operating systems and platforms using Java and have achieved linear performance improvements across machine clusters. An excellent survey of current implementations of parallel genetic algorithms can be found in Cantu-Paz [26] and Alba and Troya [30]. Although these implementations provide different kinds of distributed models, neither of them focuses on the robustness of the algorithm with respect to network failure or hardware problems such as one of the computers being removed from the computations. The approach
proposed in this document is designed from scratch keeping such issues in mind so as to provide a robust implementation of a distributed genetic algorithm.
3. DISTRIBUTED GENETIC ALGORITHM SOFTWARE

3.1. THE DISTRIBUTED GENETIC ALGORITHM MODEL

As discussed in Section 2.2, a genetic algorithm solves a problem by creating a set of solutions, pairing solutions to form newer solutions using genetic operators such as crossover and mutation, evaluating the fitness of these new solutions, and applying selection strategies to form a new set of feasible solutions. Of these tasks, the most computationally intensive task involved is the fitness function evaluation. The steps involved in evaluating a possible solution can be given as:

- Decoding the chromosome
- Calculating the fitness function value

It is evident that each chromosome can be evaluated independent of the other chromosomes in the same generation. The other tasks such as crossover and mutation can also be performed independently once pairs of chromosomes have been determined. Thus, the large task of forming a new population can be broken down into the repetitious task of manipulating pairs of chromosomes and evaluating their fitness. This theory forms the basis for the proposed distributed genetic algorithm. The entire parallelization process for a population is shown in Figure 6.
Figure 6: Parallelization of a population in a genetic algorithm.

The new population thus formed then becomes the basis for the next generation and the algorithm continues until it has evolved for the specified number of generations or the objective is met.

In order to achieve parallelization, the proposed software employs client-server architecture shown in Figure 7.
Figure 7: The client-server architecture of the genetic algorithm.

The server maintains the population of chromosomes. The clients connect to the server and request chromosomes. The client maintains a pool or buffer of chromosomes which ensures that the client is never idle. A separate thread sequentially accesses this buffer and selects a pair of chromosomes, which then undergo crossover and mutation. The resultant offspring is evaluated and sent back to the server and stored in a separate buffer on the server. Once the buffer of offspring chromosomes reaches a certain limit, the server employs selection strategies to select chromosomes that will form the new population. Figure 8 shows the flowchart for the operation of the server and the client in the genetic algorithm and the software implementation and operation are discussed in Appendix A - Software Implementation.
Figure 8: Flow-chart describing the operation of the client and the server.
3. 2. GENETIC ALGORITHM FEATURES

3. 2. 1. CHROMOSOME REPRESENTATION

The representation of a chromosome follows the genetic representation as given by Cheng, Gen and Tsujimura [31] where each gene in a chromosome represents a job and its position in the chromosome refers to the operation to be performed. The length of the chromosome will be the sum of all the operations on all jobs. Table 2 shows one of the benchmark problems used in testing the genetic algorithm. For such a scenario, the encoding of the chromosome is given in Table 3.

<table>
<thead>
<tr>
<th>Job</th>
<th>Operation (machine, time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3, 1 1, 3 2, 6 4, 7 6, 3 5, 6</td>
</tr>
<tr>
<td>2</td>
<td>2, 8 3, 5 5, 10 6, 10 1, 10 4, 4</td>
</tr>
<tr>
<td>3</td>
<td>3, 5 4, 4 6, 8 1, 9 2, 1 5, 7</td>
</tr>
<tr>
<td>4</td>
<td>2, 5 1, 5 3, 5 4, 3 5, 8 6, 9</td>
</tr>
<tr>
<td>5</td>
<td>3, 9 2, 3 5, 5 6, 4 1, 3 4, 1</td>
</tr>
<tr>
<td>6</td>
<td>2, 3 4, 3 6, 9 1, 10 5, 4 3, 1</td>
</tr>
</tbody>
</table>

Table 2: The Fisher and Thompson 6x6 instance (ft06).
(Source: OR Library, Imperial College Management School, London [34])

<table>
<thead>
<tr>
<th>Job</th>
<th>6 1 4 4 6 2 3 3 1 1 2 1 5 5 2 2 4 6 6 6 3 5 2 3 1 5 2 4 4 2 6 1 4 3 3 5 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>1 1 1 2 2 1 1 2 2 3 2 4 1 2 3 3 3 4 5 3 3 4 4 5 4 5 4 5 6 6 6 5 6 5 6</td>
</tr>
</tbody>
</table>
3.2.2. Genetic Operators

The genetic algorithm has a random number generator that generates numbers between 0 and 1 for every pair of chromosomes. If the value of this number is less than or equal to the crossover probability specified for the algorithm, the selected chromosomes undergo crossover. The genetic algorithm uses order-based crossover as proposed by Davis [32], where a substring is selected from one parent at random and copied to the offspring in their respective positions. The genes selected from the first parent are then removed from the second parent and the remaining genes are copied into the blank positions of the offspring chromosome. Consider two parent chromosomes $C_1$ and $C_2$ undergoing crossover as shown in Figure 9. For a chromosome of length 12, there are 11 cutting points and the probability of each cutting point in the chromosome is given as:

$$p = \frac{\text{position of cutting point}}{\text{total number of cut points}} = \frac{i}{11}$$

<table>
<thead>
<tr>
<th>C1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>3</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i →</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Crossover point</td>
<td>0.0949</td>
<td>0.1818</td>
<td>0.2727</td>
<td>0.3636</td>
<td>0.4545</td>
<td>0.5455</td>
<td>0.6364</td>
<td>0.7273</td>
<td>0.8182</td>
<td>0.9090</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 9: Cutting point probabilities in a chromosome.**
A random number generator then provides us the cutting point in the first parent chromosome. In the next step, shown in Figure 10, the genes before the cutting point in \( C_1 \) are copied over to offspring \( O_1 \).

![Figure 10: Transferring genes from the first parent to the offspring.](image)

In order to maintain the validity of the chromosome, the genes that have been transferred from \( C_1 \) to \( O_1 \) are then removed from \( C_2 \) starting from the end of the chromosome, as shown in Figure 11.
Finally, the remaining genes are copied over to O₁, while maintaining the order in C₂ as shown in Figure 12.
After crossover, another random number between 0 and 1 is generated for the resultant chromosome. If this number is less than or equal to the mutation probability, this chromosome undergoes mutation. Generally, mutation occurs less frequently compared to crossover because mutation probability is very low while crossover probability is very high. If mutation occurs, the offspring chromosome then undergoes mutation using a pair exchange mutation operator as specified in Gen, Tsujimura and Kubota [33]. As shown in Figure 13, two non-identical jobs are selected based on random numbers, and are swapped to create a new offspring.

![Figure 13: Swapping two non-identical genes in a chromosome.](image)

3. 2. 3. EVALUATION FUNCTION

The evaluation function is a measure of the fitness and quality of a chromosome relative to the other chromosomes in a population. For a job shop, the makespan is a good way to determine machine utilization. The evaluation function in the genetic algorithm is based on a first-in first-out (FIFO) schedule builder. The scheduler calculates the makespan based on the information given in the chromosome and the scenario. Figure
14 shows the Gantt chart for the makespan for the chromosome shown in Table 3 for the scenario given in Table 2.

![Gantt chart](image)

**Figure 14: Makespan calculation of a chromosome using FIFO builder.**

3.2.4. **SELECTION STRATEGIES**

In order to form the population, the genetic algorithm employs selection strategies to select chromosomes from the buffer of processed chromosomes. The proposed genetic algorithm uses a roulette wheel based selection strategy. The chromosomes are sorted in ascending order of their evaluation function. Reproduction probabilities are assigned to each chromosome based on the overall quality of the entire population. A random number generator then decides the chromosomes that are carried over to the next generation. Consider a sample population of 10 chromosomes shown in Table 4. These chromosomes are then sorted and reproduction probabilities are calculated as shown in Table 5.
Table 4: Chromosomes in a sample population.

<table>
<thead>
<tr>
<th>No.</th>
<th>Chromosome</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C₁</td>
<td>59</td>
</tr>
<tr>
<td>2</td>
<td>C₂</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>C₃</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td>C₄</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>C₅</td>
<td>66</td>
</tr>
<tr>
<td>6</td>
<td>C₆</td>
<td>58</td>
</tr>
<tr>
<td>7</td>
<td>C₇</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>C₈</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>C₉</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>C₁₀</td>
<td>67</td>
</tr>
</tbody>
</table>

Once the chromosomes have been sorted as shown in Table 5, their fitness values are summed up and probabilities are assigned to each chromosome. Since this is a minimization problem, probabilities are inverted by dividing by the sum of all probabilities so that chromosomes with lower fitness values will have higher probability of being carried over for the next generation.

Table 5: Calculation of reproduction probabilities of chromosomes.

<table>
<thead>
<tr>
<th>No.</th>
<th>Chromosome</th>
<th>Value</th>
<th>Probability $p_i = \frac{\sum v - v_i}{\sum v}$</th>
<th>Reproduction Probability $p_{ri} = \frac{p_i}{\sum p}$</th>
<th>Range $r_i = p_{ri} + p_{ri-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C₃</td>
<td>55</td>
<td>0.913522</td>
<td>0.101502</td>
<td>0.101502</td>
</tr>
<tr>
<td>2</td>
<td>C₇</td>
<td>55</td>
<td>0.913522</td>
<td>0.101502</td>
<td>0.203005</td>
</tr>
<tr>
<td>3</td>
<td>C₆</td>
<td>58</td>
<td>0.908805</td>
<td>0.100978</td>
<td>0.303983</td>
</tr>
<tr>
<td>4</td>
<td>C₁</td>
<td>59</td>
<td>0.907233</td>
<td>0.100804</td>
<td>0.404787</td>
</tr>
<tr>
<td>5</td>
<td>C₈</td>
<td>60</td>
<td>0.90566</td>
<td>0.100629</td>
<td>0.505416</td>
</tr>
<tr>
<td>6</td>
<td>C₂</td>
<td>64</td>
<td>0.899371</td>
<td>0.09993</td>
<td>0.605346</td>
</tr>
<tr>
<td>7</td>
<td>C₅</td>
<td>66</td>
<td>0.896226</td>
<td>0.09935</td>
<td>0.704927</td>
</tr>
<tr>
<td>8</td>
<td>C₁₀</td>
<td>67</td>
<td>0.894654</td>
<td>0.099406</td>
<td>0.804333</td>
</tr>
<tr>
<td>9</td>
<td>C₄</td>
<td>72</td>
<td>0.886792</td>
<td>0.098532</td>
<td>0.902865</td>
</tr>
<tr>
<td>10</td>
<td>C₉</td>
<td>80</td>
<td>0.874214</td>
<td>0.097135</td>
<td>1.000000</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>636</td>
<td>9.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A random number generator is used to generate random numbers between 0 and 1. If the random number generated is 0.0944, chromosome $C_3$ is selected, and if the number 0.2324 is generated, chromosome $C_6$ is selected, and so on, until the required chromosomes have been selected for the next generation.
4. TESTING AND RESULTS

4.1. APPROACH

The genetic algorithm implementation was tested in several environments to determine the effectiveness of the distributed architecture. Testing was also done to determine robustness of the server in recovering from client disconnections. Timed tests were performed in three different scenarios. Tests were also conducted to determine whether increasing the number of clients would affect the efficiency of the genetic algorithm.

- Case 1: Identical computers with 10 MBPS LAN and external network traffic.
- Case 2: Identical computers with 1000 MBPS LAN, isolated from external traffic.
- Case 3: Non-identical computers on a 100 MBPS LAN with external network traffic.
- Case 4: Identical computers with 1000 MBPS LAN, isolated from external traffic testing solution quality.

Each case was tested with different types of problems. In each case, the single client tests have been performed with the client and the server running on the same computer so as to avoid influence of network conditions on these tests and achieve performance closer to a non-distributed genetic algorithm. In all multiple client tests, a dedicated computer was used as a server and dedicated computers running client software connected to the server over the network. These tests were performed with the genetic algorithm parameters given in Table 6.
Table 6: Parameters used in testing the genetic algorithm.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>200</td>
</tr>
<tr>
<td>Generations</td>
<td>200</td>
</tr>
<tr>
<td>Crossover Rate</td>
<td>0.99</td>
</tr>
<tr>
<td>Mutation Rate</td>
<td>0.05</td>
</tr>
<tr>
<td>Number Of Mutations</td>
<td>1</td>
</tr>
<tr>
<td>Elitism Factor</td>
<td>1</td>
</tr>
</tbody>
</table>

4. 2. CASE 1

4. 2. 1. TEST CONFIGURATION

In this test, seven computers with identical hardware and software installed and a variety of processes running on them were used to test the effectiveness of the genetic algorithm. All the computers were connected to a 10 MBPS network hub. Other computers not being used in the test were also connected to the hub and the hub was connected to the internet. The genetic algorithm was tested using various numbers of clients.

The hardware that was used in performing these tests is given in Table 7.

Table 7: Hardware and software configuration for Case 1.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Dell Optiplex GX260</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Pentium4 1.8GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>384 MB SDRAM</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>40 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Microsoft Windows 2000 SP3</td>
</tr>
<tr>
<td>Java VM</td>
<td>Java 1.4.2_02 from Sun Microsystems</td>
</tr>
<tr>
<td>Network</td>
<td>10 MBPS</td>
</tr>
</tbody>
</table>
The three test cases that were used in performing these tests were obtained from the Imperial College Management School [34].

- Instance ft06 – Fisher and Thompson 6 x 6 instance.
- Instance ft10 – Fisher and Thompson 10 x 10 instance.
- Instance la26 – Lawrence 20 x 10 instance.

4.2.2. RESULTS

Test results obtained during testing the genetic algorithm are given in Table 8, Table 9 and Table 10.

**Table 8: Time required for solving the 6x6 problem.**

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>607.172</td>
<td>345.328</td>
<td>367.953</td>
<td>378.094</td>
<td>804.641</td>
<td>801.875</td>
</tr>
<tr>
<td>2</td>
<td>517.891</td>
<td>345.891</td>
<td>371.483</td>
<td>382.297</td>
<td>803.687</td>
<td>804.125</td>
</tr>
<tr>
<td>3</td>
<td>515.453</td>
<td>350.187</td>
<td>371.969</td>
<td>385.281</td>
<td>803.672</td>
<td>802.907</td>
</tr>
<tr>
<td>4</td>
<td>530.078</td>
<td>350.282</td>
<td>367.579</td>
<td>378.454</td>
<td>804.594</td>
<td>800.891</td>
</tr>
<tr>
<td>5</td>
<td>523.110</td>
<td>344.375</td>
<td>368.531</td>
<td>390.922</td>
<td>803.797</td>
<td>795.844</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>538.741</strong></td>
<td><strong>347.213</strong></td>
<td><strong>369.503</strong></td>
<td><strong>383.010</strong></td>
<td><strong>804.078</strong></td>
<td><strong>801.128</strong></td>
</tr>
</tbody>
</table>
Table 9: Time required for solving the 10x10 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>Number Of Clients</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>856.750</td>
<td>548.406</td>
<td>493.703</td>
<td>485.813</td>
<td>505.922</td>
<td>516.250</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>761.500</td>
<td>541.187</td>
<td>480.875</td>
<td>489.046</td>
<td>510.110</td>
<td>516.015</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>740.907</td>
<td>544.906</td>
<td>492.890</td>
<td>488.734</td>
<td>503.296</td>
<td>517.406</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>812.047</td>
<td>540.219</td>
<td>482.469</td>
<td>482.266</td>
<td>509.485</td>
<td>516.141</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>791.762</td>
<td>538.266</td>
<td>486.750</td>
<td>485.656</td>
<td>500.203</td>
<td>519.937</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>792.593</td>
<td>542.597</td>
<td>487.337</td>
<td>486.303</td>
<td>505.803</td>
<td>517.150</td>
</tr>
</tbody>
</table>

Table 10: Time required for solving the 20x10 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>Number Of Clients</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1037.203</td>
<td>797.344</td>
<td>706.359</td>
<td>676.641</td>
<td>663.375</td>
<td>661.954</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>983.015</td>
<td>794.125</td>
<td>708.187</td>
<td>678.078</td>
<td>662.985</td>
<td>665.062</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>889.688</td>
<td>798.875</td>
<td>712.360</td>
<td>665.391</td>
<td>661.297</td>
<td>666.016</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>958.765</td>
<td>799.047</td>
<td>698.157</td>
<td>673.234</td>
<td>661.516</td>
<td>665.641</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>868.766</td>
<td>793.141</td>
<td>705.140</td>
<td>673.422</td>
<td>660.469</td>
<td>666.985</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>947.487</td>
<td>796.506</td>
<td>706.041</td>
<td>673.353</td>
<td>661.928</td>
<td>665.132</td>
</tr>
</tbody>
</table>

4.3. CASE 2

4.3.1. TEST CONFIGURATION

This test utilized new computers with the latest processors and very high memory. All the computers were connected to a 1000 MBPS network hub through gigabit Ethernet cards. The hub was not connected to the internet and no other software was installed on any of the computers.

A total of five identical machines were used in this test. Each machine had the hardware and software configuration given in Table 11.
Table 11: Hardware and software configuration for Case 2.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>Dell Optiplex SX270</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Pentium4 2.8GHz with HT Technology</td>
</tr>
<tr>
<td>RAM</td>
<td>1.0 GB Dual Channel DDR SDRAM</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>40 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Microsoft Windows XP Professional SP1a</td>
</tr>
<tr>
<td>Java VM</td>
<td>Java 1.4.2 from Sun Microsystems</td>
</tr>
<tr>
<td>Network</td>
<td>1000 MBPS</td>
</tr>
</tbody>
</table>

In this case, since extremely powerful machines were used, tests were performed on a total of six cases obtained from the Imperial College Management School [34].

- Instance ft06 – Fisher and Thompson 6 x 6 instance.
- Instance ft10 – Fisher and Thompson 10 x 10 instance.
- Instance la26 – Lawrence 20 x 10 instance.
- Instance la31 – Lawrence 30 x 10 instance.
- Instance yn1 - Yamada and Nakano 20 x 20 instance.
- Instance swv11 - Storer, Wu, and Vaccari 50 x 10 instance.

Larger problems were tackled in this case to observe the scaling of the algorithm where there was very little chance of external noise affecting the results.

4. 3. 2. RESULTS

Test results obtained during testing the genetic algorithm are given in Table 12, Table 13, Table 14, Table 15, Table 16 and Table 17.
### Table 12: Time required for solving the 6x6 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42.578</td>
<td>23.484</td>
<td>20.204</td>
<td>19.890</td>
</tr>
<tr>
<td>2</td>
<td>42.047</td>
<td>23.016</td>
<td>19.954</td>
<td>19.844</td>
</tr>
<tr>
<td>3</td>
<td>42.203</td>
<td>22.969</td>
<td>20.125</td>
<td>19.782</td>
</tr>
<tr>
<td>4</td>
<td>41.422</td>
<td>23.140</td>
<td>20.031</td>
<td>19.969</td>
</tr>
<tr>
<td>5</td>
<td>42.703</td>
<td>22.984</td>
<td>20.062</td>
<td>19.890</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>42.191</strong></td>
<td><strong>23.119</strong></td>
<td><strong>20.075</strong></td>
<td><strong>19.875</strong></td>
</tr>
</tbody>
</table>

### Table 13: Time required for solving the 10x10 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>69.313</td>
<td>33.656</td>
<td>31.094</td>
<td>31.047</td>
</tr>
<tr>
<td>2</td>
<td>69.922</td>
<td>33.453</td>
<td>31.156</td>
<td>31.062</td>
</tr>
<tr>
<td>3</td>
<td>67.297</td>
<td>33.234</td>
<td>30.906</td>
<td>31.469</td>
</tr>
<tr>
<td>4</td>
<td>68.344</td>
<td>33.438</td>
<td>31.047</td>
<td>31.218</td>
</tr>
<tr>
<td>5</td>
<td>69.203</td>
<td>33.594</td>
<td>31.093</td>
<td>31.172</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>68.816</strong></td>
<td><strong>33.475</strong></td>
<td><strong>31.059</strong></td>
<td><strong>31.194</strong></td>
</tr>
</tbody>
</table>

### Table 14: Time required for solving the 20x10 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>118.000</td>
<td>53.922</td>
<td>48.547</td>
<td>50.125</td>
</tr>
<tr>
<td>2</td>
<td>117.219</td>
<td>54.312</td>
<td>49.234</td>
<td>50.000</td>
</tr>
<tr>
<td>3</td>
<td>118.578</td>
<td>53.437</td>
<td>49.438</td>
<td>49.938</td>
</tr>
<tr>
<td>4</td>
<td>119.297</td>
<td>53.954</td>
<td>48.797</td>
<td>49.297</td>
</tr>
<tr>
<td>5</td>
<td>112.907</td>
<td>53.625</td>
<td>49.047</td>
<td>49.890</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>117.200</strong></td>
<td><strong>53.850</strong></td>
<td><strong>49.013</strong></td>
<td><strong>49.850</strong></td>
</tr>
</tbody>
</table>
Table 15: Time required for solving the 30x10 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>168.547</td>
<td>74.781</td>
<td>67.344</td>
<td>69.062</td>
</tr>
<tr>
<td>2</td>
<td>166.735</td>
<td>75.234</td>
<td>67.796</td>
<td>68.390</td>
</tr>
<tr>
<td>3</td>
<td>166.437</td>
<td>74.828</td>
<td>66.516</td>
<td>68.703</td>
</tr>
<tr>
<td>4</td>
<td>167.984</td>
<td>75.469</td>
<td>67.718</td>
<td>67.891</td>
</tr>
<tr>
<td>5</td>
<td>167.328</td>
<td>75.171</td>
<td>67.281</td>
<td>67.844</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>167.406</td>
<td>75.097</td>
<td>67.331</td>
<td>68.378</td>
</tr>
</tbody>
</table>

Table 16: Time required for solving the 20x20 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>193.703</td>
<td>96.578</td>
<td>83.969</td>
<td>86.266</td>
</tr>
<tr>
<td>2</td>
<td>190.859</td>
<td>96.000</td>
<td>85.953</td>
<td>86.594</td>
</tr>
<tr>
<td>3</td>
<td>197.781</td>
<td>96.547</td>
<td>84.844</td>
<td>86.844</td>
</tr>
<tr>
<td>4</td>
<td>193.125</td>
<td>95.750</td>
<td>84.438</td>
<td>85.984</td>
</tr>
<tr>
<td>5</td>
<td>189.609</td>
<td>95.515</td>
<td>84.828</td>
<td>85.610</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>193.015</td>
<td>96.078</td>
<td>84.806</td>
<td>86.260</td>
</tr>
</tbody>
</table>

Table 17: Time required for solving the 50x10 problem.

<table>
<thead>
<tr>
<th>Run</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>266.375</td>
<td>119.922</td>
<td>100.656</td>
<td>101.844</td>
</tr>
<tr>
<td>2</td>
<td>277.640</td>
<td>118.032</td>
<td>100.782</td>
<td>101.343</td>
</tr>
<tr>
<td>3</td>
<td>272.781</td>
<td>119.375</td>
<td>102.594</td>
<td>104.422</td>
</tr>
<tr>
<td>4</td>
<td>267.500</td>
<td>119.859</td>
<td>100.859</td>
<td>101.500</td>
</tr>
<tr>
<td>5</td>
<td>284.453</td>
<td>119.343</td>
<td>100.657</td>
<td>102.297</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>273.750</td>
<td>119.306</td>
<td>101.110</td>
<td>102.281</td>
</tr>
</tbody>
</table>
4. 4. CASE 3

4. 4. 1. TEST CONFIGURATION

In this test, a hybrid setup consisting of a very fast and powerful server and slower client computers was used to observe the effectiveness of the algorithm in common computing environments. Configuration of the server is given in Table 18 and that of the client is given in Table 19.

**Table 18: Hardware and software configuration of server for Case 3.**

<table>
<thead>
<tr>
<th>Computer</th>
<th>Dell Optiplex SX270</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Pentium4 2.8GHz with HT Technology</td>
</tr>
<tr>
<td>RAM</td>
<td>1.0 GB Dual Channel DDR SDRAM</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>40 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Microsoft Windows XP Professional SP1a</td>
</tr>
<tr>
<td>Java VM</td>
<td>Java 1.4.2 from Sun Microsystems</td>
</tr>
<tr>
<td>Network</td>
<td>1000 MBPS</td>
</tr>
</tbody>
</table>

**Table 19: Hardware and software configuration of clients for Case 3.**

<table>
<thead>
<tr>
<th>Computer</th>
<th>Dell Optiplex GX260</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Pentium4 1.8GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>384 MB SDRAM</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>40 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Microsoft Windows XP Professional SP1</td>
</tr>
<tr>
<td>Java VM</td>
<td>Java 1.4.2_02 from Sun Microsystems</td>
</tr>
<tr>
<td>Network</td>
<td>100 MBPS</td>
</tr>
</tbody>
</table>

For this case, tests were performed on a total of three cases obtained from the Imperial College Management School [34].
- Instance la31 – Lawrence 30 x 10 instance.
- Instance yn1 - Yamada and Nakano 20 x 20 instance.
- Instance swv11 - Storer, Wu, and Vaccari 50 x 10 instance.

4.4.2. RESULTS

Test results obtained during testing the genetic algorithm are given in Table 12, Table 13, Table 14, Table 15, Table 16 and Table 17. Figure 16 provides a graphical summary of the results obtained during these tests.

<p>| Table 20: Time required for solving the 30x10 problem. |</p>
<table>
<thead>
<tr>
<th>Run</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>178.046</td>
<td>116.157</td>
<td>90.031</td>
<td>81.125</td>
</tr>
<tr>
<td>2</td>
<td>180.891</td>
<td>117.953</td>
<td>90.984</td>
<td>80.032</td>
</tr>
<tr>
<td>3</td>
<td>180.203</td>
<td>114.500</td>
<td>92.109</td>
<td>81.687</td>
</tr>
<tr>
<td>4</td>
<td>180.000</td>
<td>116.719</td>
<td>91.328</td>
<td>81.687</td>
</tr>
<tr>
<td>5</td>
<td>184.281</td>
<td>115.531</td>
<td>91.183</td>
<td>81.047</td>
</tr>
<tr>
<td>Average</td>
<td>180.684</td>
<td>116.172</td>
<td>91.127</td>
<td>81.075</td>
</tr>
</tbody>
</table>

<p>| Table 21: Time required for solving the 20x20 problem. |</p>
<table>
<thead>
<tr>
<th>Run</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>209.390</td>
<td>137.344</td>
<td>109.687</td>
<td>97.813</td>
</tr>
<tr>
<td>2</td>
<td>211.453</td>
<td>137.391</td>
<td>107.719</td>
<td>97.907</td>
</tr>
<tr>
<td>3</td>
<td>211.485</td>
<td>139.063</td>
<td>109.406</td>
<td>96.828</td>
</tr>
<tr>
<td>4</td>
<td>213.469</td>
<td>135.141</td>
<td>108.469</td>
<td>97.125</td>
</tr>
<tr>
<td>5</td>
<td>210.782</td>
<td>140.125</td>
<td>109.391</td>
<td>97.078</td>
</tr>
<tr>
<td>Average</td>
<td>211.316</td>
<td>137.813</td>
<td>108.934</td>
<td>97.350</td>
</tr>
</tbody>
</table>
Table 22: Time required for solving the 50x10 problem.

<table>
<thead>
<tr>
<th></th>
<th>Number Of Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>242.343</td>
</tr>
<tr>
<td>2</td>
<td>245.390</td>
</tr>
<tr>
<td>3</td>
<td>239.532</td>
</tr>
<tr>
<td>4</td>
<td>238.250</td>
</tr>
<tr>
<td>5</td>
<td>243.234</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>241.750</td>
</tr>
</tbody>
</table>

4. 5. Case 4

4. 5. 1. Test Configuration

The hardware and software used during these tests was the same as used in performing tests for Case 2 and these specifications are given in Table 11. Only one problem was tested in this case but tests were performed over a larger number of runs. The problem data was obtained from Imperial College Management School [34].

- Instance ft10 – Fisher and Thompson 10 x 10 instance.

Results of these tests were recorded to determine the time required to solve the problem and the best makespan recorded in each test.
4.5.2. **RESULTS**

The time required for solving the problem for different number of clients is given in Table 23. The best makespan recorded for each run is given in Table 24.

**Table 23 : Time required for solving the 10x10 problem.**

<table>
<thead>
<tr>
<th>Run</th>
<th>Number Of Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>167.281 83.219 75.922 76.969</td>
</tr>
<tr>
<td>2</td>
<td>164.703 82.986 76.140 75.984</td>
</tr>
<tr>
<td>3</td>
<td>162.406 82.406 76.203 76.391</td>
</tr>
<tr>
<td>4</td>
<td>165.204 83.172 75.906 76.984</td>
</tr>
<tr>
<td>5</td>
<td>157.453 82.203 75.078 76.687</td>
</tr>
<tr>
<td>6</td>
<td>169.875 83.203 75.641 76.078</td>
</tr>
<tr>
<td>7</td>
<td>165.141 82.547 76.781 77.359</td>
</tr>
<tr>
<td>8</td>
<td>160.969 82.187 75.968 77.078</td>
</tr>
<tr>
<td>9</td>
<td>166.109 82.422 75.687 76.500</td>
</tr>
<tr>
<td>10</td>
<td>156.906 82.500 75.640 76.843</td>
</tr>
<tr>
<td>11</td>
<td>162.437 82.828 75.829 76.907</td>
</tr>
<tr>
<td>12</td>
<td>160.453 82.188 76.078 76.171</td>
</tr>
<tr>
<td>13</td>
<td>161.656 82.875 75.703 76.062</td>
</tr>
<tr>
<td>14</td>
<td>160.985 84.359 75.641 76.437</td>
</tr>
<tr>
<td>15</td>
<td>166.329 82.422 75.922 77.110</td>
</tr>
<tr>
<td>16</td>
<td>162.265 82.563 76.219 76.750</td>
</tr>
<tr>
<td>17</td>
<td>162.266 82.016 75.547 76.734</td>
</tr>
<tr>
<td>18</td>
<td>161.344 82.688 75.734 76.281</td>
</tr>
<tr>
<td>19</td>
<td>158.671 81.875 75.734 76.844</td>
</tr>
<tr>
<td>20</td>
<td>161.703 82.547 75.640 77.218</td>
</tr>
</tbody>
</table>

**Average** 162.708 82.660 75.851 76.669
Table 24: Makespan for the 10x10 problem for different number of clients.

<table>
<thead>
<tr>
<th>Run</th>
<th>Number Of Clients</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1105</td>
<td>1113</td>
<td>1096</td>
<td>1153</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1094</td>
<td>1103</td>
<td>1031</td>
<td>1130</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1147</td>
<td>1046</td>
<td>1209</td>
<td>1148</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1123</td>
<td>1127</td>
<td>1122</td>
<td>1098</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1122</td>
<td>1183</td>
<td>1170</td>
<td>1099</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1148</td>
<td>1105</td>
<td>1124</td>
<td>1045</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1097</td>
<td>1188</td>
<td>1194</td>
<td>1063</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1237</td>
<td>1089</td>
<td>1074</td>
<td>1101</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1052</td>
<td>1137</td>
<td>1141</td>
<td>1087</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1119</td>
<td>1144</td>
<td>1083</td>
<td>1064</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>1042</td>
<td>1023</td>
<td>1189</td>
<td>1110</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>1139</td>
<td>1080</td>
<td>1084</td>
<td>1088</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>1113</td>
<td>1127</td>
<td>1124</td>
<td>1125</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>1123</td>
<td>1131</td>
<td>1104</td>
<td>1224</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>1096</td>
<td>1071</td>
<td>1127</td>
<td>1190</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>1123</td>
<td>1169</td>
<td>1156</td>
<td>1144</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>1144</td>
<td>1072</td>
<td>1115</td>
<td>1069</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>1215</td>
<td>1162</td>
<td>1100</td>
<td>1096</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>1101</td>
<td>1079</td>
<td>1105</td>
<td>1113</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>1172</td>
<td>1088</td>
<td>1054</td>
<td>1110</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>1125.60</td>
<td>1111.85</td>
<td>1120.10</td>
<td>1112.85</td>
</tr>
</tbody>
</table>
5. ANALYSIS AND CONCLUSIONS

5.1. ANALYSIS OF RESULTS

A graphical summary of results of testing for cases 1, 2 and 3 has been given in Figure 15, Figure 16 and Figure 17 respectively.

![Graph: Time vs Number Of Clients](image)

**Figure 15:** Summary of test results obtained in Case 1.
Figure 16: Summary of test results obtained in Case 2.

Figure 17: Summary of test results obtained in Case 3.
All of these results are indicative of the effectiveness of the proposed distributed genetic algorithm model. Since Case 1 was performed using older hardware with slower network connectivity, the processing times observed are much higher as compared to the other two cases.

In Case 1, the 6x6 problem solution shows an interesting curve. During testing, it was observed that for such smaller problems, the server could not cope with the large number of clients accessing it at extremely short intervals to return processed chromosomes. The server was choked with client requests and hence, after 2 clients, the solution time kept on increasing when solving the 6x6 problem. Similar effect was observed in the case of the 10x10 problem but the choking occurred for a higher number of clients as compared to the 6x6 problem. This indicated that distributing a small problem to several clients would not be very effective in reducing the solution time. However, for larger problems, distributing the problem did prove effective and it would be a matter of simple testing to determine the optimal number of clients that would be required to solve a large problem effectively using the distributed model.

In Case 2, very high end hardware was used over an extremely fast network. The results obtained indicate that when moving from a single client to two clients almost halves the solution time. However, the gain in going from two to three clients is not very high in case of the smaller problems. This supports the earlier conclusion that it may not be very beneficial to apply distributed computing when solving small problems.
In Case 3, the tests were conducted using an extremely powerful server and slower client machines to ensure that the server was not choked with client requests. Tests results indicate that in this case, the effect of choking was not observed as the high speed server could effectively manage the clients. In each problem, the solution time decreased as more clients were added.

A graphical summary of the timed tests for Case 4 is given in Figure 18 and the makespan obtained in each run is given in Figure 19.

Figure 18: Summary of timed test results obtained in Case 4.
One-way analysis of variance (ANOVA) was performed on the makespan results given in Table 24 and the results of the analysis are given in Table 25:
Table 25: One-way ANOVA results for makespan data.

<table>
<thead>
<tr>
<th>Groups</th>
<th>Count</th>
<th>Sum</th>
<th>Average</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Client</td>
<td>20</td>
<td>22512</td>
<td>1125.60</td>
<td>2126.174</td>
</tr>
<tr>
<td>2 Clients</td>
<td>20</td>
<td>22237</td>
<td>1111.85</td>
<td>1996.450</td>
</tr>
<tr>
<td>3 Clients</td>
<td>20</td>
<td>22402</td>
<td>1120.10</td>
<td>2158.936</td>
</tr>
<tr>
<td>4 Clients</td>
<td>20</td>
<td>22257</td>
<td>1112.85</td>
<td>1900.134</td>
</tr>
</tbody>
</table>

ANOVA

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
<th>F</th>
<th>P-value</th>
<th>F crit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between Groups</td>
<td>2517.50</td>
<td>3.00</td>
<td>839.166</td>
<td>0.4103</td>
<td>0.7461</td>
<td>2.7249</td>
</tr>
<tr>
<td>Within Groups</td>
<td>155451.70</td>
<td>76.00</td>
<td>2045.417</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>157969.20</td>
<td>79.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the ANOVA results, the P-value obtained is 0.7461. Such a high value leads to the inference that the data in each column belongs to the same group. Therefore, the quality of solution, measured by makespan, can be assumed to be independent of the number of processors which were used to solve the problem.

5.2. Conclusions

This research concludes that the proposed distributed genetic algorithm can be effective in improving solution times for large job shop scheduling problems. ANOVA results obtained by analyzing makespan values obtained for different number of client machines indicates that the proposed algorithm reduces the problem solving time while still retaining the efficiency associated with genetic algorithms. Thus, the proposed model would be ideal in solving extremely large job shop scheduling problems in a short amount of time while getting good results.
It is evident that linear gain was not achieved in any case and this can be attributed to several factors, but the most important factor is the overhead that each client adds to the server and the network. Other factors include network related issues as well as the overhead of different services running on each client.

5.3. FUTURE RESEARCH

Several avenues exist for future research in this area. This algorithm can further be tested on even larger problems with more clients connecting over varying network speeds. Since the algorithm has been implemented in Java and the source code will be made available for further research, it is possible to build upon the existing framework and use it to solve other problems. The proposed genetic algorithm model can be applied to try and solve other large optimization problems. It is also possible to look into ways to make the server more intelligent so that it determines the optimal number of clients required in solving a problem and refuses to accept client connections beyond that number.
REFERENCES


APPENDIX A - SOFTWARE IMPLEMENTATION

Software for the genetic algorithm has been implemented using Java 1.4.2. The software consists of a primary framework of interfaces that are implemented to define the functionality of the genetic algorithm. Figure 20 shows the UML diagram for the edu.ohiou.ga.interfaces package.
Figure 20: UML Diagram of the *edu.ohiou.ga.interfaces* package.
The functionality of the server is defined by the interface `Server`. The package `edu.ohiou.ga.server` provides the implementation of the interface to create a genetic algorithm server that manages the connections and the population. Figure 21 shows the UML structure of the `edu.ohiou.ga.server` package.
Figure 21: UML Diagram of the edu.ohiou.ga.server package.
The server manages client connections by storing the list of connected clients in a map. Each client’s connection to the server is verified periodically to ensure that all the connections are valid. If any broken connections are found, the disconnected clients are removed from the map. The genetic algorithm relies on the `edu.ohiou.ga.jobshop` package to solve the job shop scheduling problem. Figure 22 shows the UML structure of the `edu.ohiou.ga.jobshop` package.
Figure 22: UML diagram for the edu.ohiou.ga.jobshop package.
On startup, the server reads an input XML file that defines the problem to be solved. The XML file for the job shop problem is based on the Document Type Definition (DTD) given in Figure 23 and the XML file for the ft06 (6 x 6) problem is given in Figure 24.

```xml
<!DOCTYPE Solution [
<!ELEMENT Scenario (Job*)>
<!ATTLIST Scenario
   ScenarioID CDATA #REQUIRED
   OptimalSolution CDATA #REQUIRED
   Jobs CDATA #REQUIRED
   Machines CDATA #REQUIRED
>]

<!ELEMENT Job (Operation*)>
<!ATTLIST Job
   JobID CDATA #REQUIRED
   DueDate CDATA #REQUIRED
>

<!ELEMENT Operation EMPTY>
<!ATTLIST Operation
   OperationID CDATA #REQUIRED
   MachineID CDATA #REQUIRED
   Time CDATA #REQUIRED
>
]
```

**Figure 23: DTD for the input XML file that stores the problem data.**
<?xml version="1.0" encoding="ASCII" ?>
<!DOCTYPE Solution (View Source for full doctype ...)>  
  <Scenario ScenarioID="ft06" OptimalSolution="55" Jobs="6" Machines="6">  
    - <Job JobID="1">  
      <Operation OperationID="1" MachineID="3" Time="1" />  
      <Operation OperationId="2" MachineID="2" Time="8" />  
      <Operation OperationId="3" MachineID="3" Time="5" />  
      <Operation OperationId="4" MachineID="2" Time="5" />  
      <Operation OperationId="5" MachineID="3" Time="9" />  
      <Operation OperationId="6" MachineID="2" Time="3" />  
    </Job>  
    - <Job JobID="2">  
      <Operation OperationId="1" MachineID="1" Time="3" />  
      <Operation OperationId="2" MachineID="3" Time="5" />  
      <Operation OperationId="3" MachineID="4" Time="4" />  
      <Operation OperationId="4" MachineID="1" Time="5" />  
      <Operation OperationId="5" MachineID="2" Time="3" />  
      <Operation OperationId="6" MachineID="4" Time="3" />  
    </Job>  
    - <Job JobID="3">  
      <Operation OperationId="1" MachineID="2" Time="6" />  
      <Operation OperationId="2" MachineID="5" Time="10" />  
      <Operation OperationId="3" MachineID="6" Time="8" />  
      <Operation OperationId="4" MachineID="3" Time="5" />  
      <Operation OperationId="5" MachineID="5" Time="5" />  
      <Operation OperationId="6" MachineID="6" Time="9" />  
    </Job>  
    - <Job JobID="4">  
      <Operation OperationId="1" MachineID="4" Time="7" />  
      <Operation OperationId="2" MachineID="6" Time="10" />  
      <Operation OperationId="3" MachineID="1" Time="9" />  
      <Operation OperationId="4" MachineID="4" Time="3" />  
      <Operation OperationId="5" MachineID="6" Time="4" />  
      <Operation OperationId="6" MachineID="1" Time="10" />  
    </Job>  
    - <Job JobID="5">  
      <Operation OperationId="1" MachineID="6" Time="3" />  
      <Operation OperationId="2" MachineID="1" Time="10" />  
      <Operation OperationId="3" MachineID="2" Time="1" />  
      <Operation OperationId="4" MachineID="5" Time="8" />  
      <Operation OperationId="5" MachineID="1" Time="3" />  
      <Operation OperationId="6" MachineID="5" Time="4" />  
    </Job>  
    - <Job JobID="6">  
      <Operation OperationId="1" MachineID="5" Time="6" />  
      <Operation OperationId="2" MachineID="4" Time="4" />  
      <Operation OperationId="3" MachineID="5" Time="7" />  
      <Operation OperationId="4" MachineID="6" Time="9" />  
      <Operation OperationId="5" MachineID="4" Time="1" />  
      <Operation OperationId="6" MachineID="3" Time="1" />  
    </Job>  
  </Scenario>

Figure 24: XML file for the ft06 (6 x 6) problem.
Once the file is read and the appropriate data structures are created, the server creates an initial population of random chromosomes based on the scenario information. At this point the server is ready to accept incoming connections and starts waiting.

The client framework is provided by the `edu.ohiou.ga.client` package. Each client is a Java RMI server that connects to the genetic algorithm server and starts processing chromosomes. The UML structure of the `edu.ohiou.ga.client` package is given in Figure 25.
Figure 25: UML structure of the `edu.ohiou.ga.client` package.
Upon connection, the client is registered to the server and the scenario information is passed on from the server to the client. The client then spawns a thread that monitors the buffer of chromosomes maintained by it. This buffer is continuously filled with chromosomes fetched from the server. If the buffer is full, this thread remains idle. The client removes pairs of chromosomes from this buffer and performs crossover on them. The resultant chromosome undergoes mutation and the offspring is then evaluated based on the fitness function (makespan). Once the fitness has been calculated, the chromosome is returned to the server and stored in a separate buffer. The server monitors this buffer and once it reaches the desired size, clients are paused. The processed chromosomes are then selected on the basis of reproduction probabilities for creating the next generation. Once pairs have been selected, clients are notified to start processing again. At the end of the specified number of generations, the clients are notified to shut down and the server prints out the best and the worst solutions on the screen.

Figure 26 shows the genetic algorithm server in operation with one client connected to the server.
Figure 26: The genetic algorithm server in operation.

A screen shot of the genetic client connected to the server and processing chromosomes is given in Figure 27.

Figure 27: The genetic algorithm client in operation.
Figure 28 shows the screen shot of the genetic algorithm server at the end of the cycle. The best and the worst chromosome for the entire cycle are printed out at the end of the cycle.

Figure 28: The genetic algorithm server at the end of the GA cycle.
package edu.ohiou.ga.client;

/**
 * A simple thread that monitors the buffer size of the client and replenishes
 * it by requesting chromosomes from the server.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;
import java.rmi.RemoteException;

import edu.ohiou.ga.interfaces.Chromosome;
import edu.ohiou.ga.interfaces.Server;

public class ClientBufferMonitor extends Thread implements Serializable
{
    /**
     * The handle to the client
     */
    private ClientEngine client;

    /**
     * The handle to the server
     */
    private Server server;

    /**
     * Constructs a buffer monitor thread that will continuously monitor the
     * client's buffer and replenish it with chromosomes from the server.
     * @param client The genetic algorithm client that this thread belongs to.
     */
    public ClientBufferMonitor(ClientEngine client)
    {
        this.client = client;
        server = client.getServer();
    }

    /**
     * Monitors the client buffer and adds chromosomes to the client by
     * requesting them from the server.
     */
    public void run()
    {
        while (client.runClient())
        {
            try
            {
Chromosome[] ch = server.getChromosomePair();
client.addChromosomeToBuffer(ch[0]);
client.addChromosomeToBuffer(ch[1]);
} catch (RemoteException ex) {
    ex.printStackTrace();
} 
} //EndOfClass ClientBufferMonitor
package edu.ohiou.ga.client;

/**
 * Main program that controls the client.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.File;
import java.net.*;
import edu.ohiou.ga.GaException;
import edu.ohiou.ga.GaParameters;
import edu.ohiou.ga.interfaces.*;
import edu.ohiou.ga.jobshop.*;

public class ClientControl
{
    public static void main(String[] args)
    {
        byte[] ip = new byte[4];
        ClientEngine client = null;
        InetAddress host = null;
        try
        {
            double cProb = 0.0; //crossover probability
            double mProb = 0.0; //mutation probability
            int i = 0;
            while (i < args.length)
            {
                if (args[i].startsWith("-"))
                {
                    char flag = args[i].charAt(1);
                    switch (flag)
                    {
                        case 's':
                            //server name
                            ip = Inet4Address.getByName(
                                args[++i]).getAddress();
                            break;
                        case 'c':
                            //crossover probability
                            cProb = Double.parseDouble(args[++i]);
                            break;
                        case 'm':
                            //mutation probability
                            mProb = Double.parseDouble(args[++i]);
                            break;
                        default:
                            break;
                    }
                } else { ip[i] = args[i].charAt(0); }
            }
        }
    }
}
default:
    System.out.println("Invalid option: "+flag);
    break;
}
}
i++;
}

Mutation mut = new JSPOrderMutation(mProb, 1);
Crossover crs = new JSPOrderCrossover(cProb);

client = new ClientEngine(ip);
client.setCrossover(crs);
client.setMutation(mut);

client.startClient();
client.connect();
}

//EndOfClass ClientControl
package edu.ohiou.ga.client;

/**
 * Client implementation of the genetic algorithm. The client maintains a
 * buffer which is filled by chromosomes from the server. A separate thread
 * continuously takes out chromosomes from this buffer and processes them and
 * sends them back to the server.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;
import java.net.*;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.LinkedList;
import java.util.List;
import edu.ohiou.ga.interfaces.*;

public class ClientEngine extends UnicastRemoteObject
    implements Client, Runnable, Serializable
{

    /**
     * The buffer that holds chromosomes from the client.
     */
    private List chromosomeBuffer = new LinkedList();

    /**
     * The maximum buffer size to be maintained.
     */
    public static int MAX_BUFFER_SIZE = 20;

    /**
     * The minimum buffer size to be maintained.
     */
    public static int MIN_BUFFER_SIZE = 4;

    /**
     * Thread that runs the client operation.
     */
    private ClientThread client = null;

    /**
     * The thread that monitors the client's buffer.
     */
    private Thread bufferMonitor = null;

    /**
     * The handle to the genetic algorithm server.
private Server server;

private boolean runClient = false;

private boolean connected = false;

private Crossover crossover;

private Mutation mutation;

private Scenario scenario;

private String serverName;

private byte[] serverAddress;

public ClientEngine(byte[] serverIP) throws Exception {
    client = new ClientThread(this);
    client.setName("ClientEngine");
    serverAddress = serverIP;
    try {
        serverName = Inet4Address.getByAddress(serverIP).getHostName();
        serverName = "//" + serverName + "/Server";
    }
    catch (UnknownHostException ex) {
        ex.printStackTrace();
    }

    public void startClient() throws Exception {
    }
String clientName;

try {
    clientName = InetAddress.getLocalHost().getHostName();
    System.out.println("clientName = " + clientName);
} catch (UnknownHostException ex) {
    System.out.println("Host name not resolved... "+ "Setting default value as \'localhost\'");
    clientName = "localhost";
}

clientName = "/" + clientName + "/Client";
System.out.println("-- \" + clientName);
Naming.rebind(clientName, this);

/**
 * Set the ip address of the server to the given ip address.
 * @param serverIP the new server ip address.
 */
public void setServerAddress(byte[] serverIP) {
    serverAddress = serverIP;
}

/**
 * Returns the ip address of the server.
 * @return the ip address of the server.
 */
public byte[] getServerAddress() {
    return serverAddress;
}

/**
 * Connect to the genetic algorithm server.
 * @throws Exception if the client cannot connect to the server.
 */
public void connect() throws Exception {
    System.out.println("ClientEngine: connect()");
    server = (Server) Naming.lookup(serverName);
    server.registerClient(this);
    connected = true;
}

/**
 * Disconnect the client from the server
 * @throws Exception if the client cannot disconnect
 */
public void disconnect() throws Exception {
    System.out.println("ClientEngine: disconnect()");
if (stopClientOperation())
{
    server.removeClient(this);
    connected = false;
}

/**
 * Allows the server to remotely start the client operation.
 * @return true if the client is started successfully.
 * @throws RemoteException if the client cannot be started.
 */
public boolean startClientOperation() throws RemoteException
{
    System.out.println("ClientEngine: startClientOperation()");
    try
    {
        runClient = true;
        if (bufferMonitor == null)
        {
            bufferMonitor = new ClientBufferMonitor(this);
        }
        bufferMonitor.start();
        client.start();
        return true;
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        return false;
    }
}

/**
 * Allows the server to remotely stop the client operation.
 * @return true if the client is stopped successfully.
 * @throws RemoteException if the client operation cannot be stopped
 */
public boolean stopClientOperation() throws RemoteException
{
    System.out.println("ClientEngine: stopClientOperation()");
    runClient = false;
    try
    {
        bufferMonitor.join();
        client.join();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
    clearBuffer();
    return true;
}

/**
 * Allows the server to remotely shutdown the client.
 * @throws RemoteException if the client cannot be shutdown
 */
public void shutDownClient() throws RemoteException
{
    System.out.println("ClientEngine: shutDownClient()");
    System.exit(0);
}

/**
 * Implements the <code>Runnable</code> interface. This thread requests
 * chromosomes from the client buffer, processes them and returns them
 * to the server.
 */
public void run()
{
    System.out.println("ClientEngine: run()");
    while (runClient)
    {
        if (getBufferSize() > 4)
        {
            Chromosome ch1 = getChromosome();
            Chromosome ch2 = getChromosome();
            Chromosome res = null;
            try
            {
                res = mutation.doMutation(crossover.doCrossover(ch1, ch2));
                res.setFitness(scenario.evaluateChromosome(res));
                server.addChromosome(res);
            }
            catch (Exception ex)
            {
                System.out.println(ex.getMessage());
                ex.printStackTrace(System.out);
                System.exit(-1);
            }
        }
    }
}

/**
 * Checks whether the client buffer has chromosomes and if possible,
 * removes the first chromosome from the buffer and returns it.
 * @return The first chromosome in the buffer.
 * @throws GaException If the client buffer is empty.
 */
public Chromosome getChromosome()
{
    Chromosome ch;
    synchronized (chromosomeBuffer)
    {
        ch = (Chromosome) chromosomeBuffer.remove(0);
    }
    synchronized (bufferMonitor)
    {
        if (chromosomeBuffer.size() <= MIN_BUFFER_SIZE)
        {
            bufferMonitor.notify();
        }
    }
    return ch;
/**
 * Clears the buffer maintained by the client. This API is used at the end
 * of every generation to make sure that the client maintains the
 * chromosomes of the current generation only.
 * @throws RemoteException if the buffer cannot be cleared.
 */
public void clearBuffer() throws RemoteException
{
    System.out.println("ClientEngine: clearBuffer()");
    chromosomeBuffer.clear();
}

/**
 * Returns the status of the client.
 * @return true if the client is to keep running, false if the client is to
 * stop operation.
 */
public boolean runClient()
{
    return runClient;
}

/**
 * Returns a handle to the genetic algorithm server.
 * @return The handle to the <code>Server</code>
 */
public Server getServer()
{
    return server;
}

/**
 * Returns the buffer size of the client
 * @return the size of the buffer
 */
public int getBufferSize()
{
    return chromosomeBuffer.size();
}

/**
 * Adds a chromosome to the client's buffer. If the size of the buffer
 * increases beyond the maximum buffer size, the monitor thread is paused.
 * @param ch The chromosome to be added to the buffer.
 */
public void addChromosomeToBuffer(Chromosome ch)
{
    synchronized (chromosomeBuffer)
    {
        chromosomeBuffer.add(ch);
    }
    synchronized (bufferMonitor)
    {
        //Check the buffer size to see whether it exceeds the maximum
        //limit. If the buffer is full, pause the monitor thread.
        if (chromosomeBuffer.size() > MAX_BUFFER_SIZE)
        {
            try
            {
            }
            catch (InterruptedException e)
            {
                // Handle interruption
            }
            // Other exceptions...
        } // End synchronization
    } // End synchronization
} // End method
{  
  bufferMonitor.wait();
}

catch (InterruptedException ex)
{
  ex.printStackTrace();
}
}

/**
 * Provides a string representation of the client. This contains the host
 * address of the client and is of the form <code>//xxx.xxx.xxx.xxx/CLIENT</code>
 */
public String getName()
{
  try
  {
    String name = InetAddress.getLocalHost().getHostName();
    return "//" + name + "/Client";
  }
  catch (UnknownHostException ex)
  {
    ex.printStackTrace();
    return null;
  }
}

/**
 * Over rides the <code>toString()</code> method to provide a string
 * representation of the client. Calls <code>getName()</code> internally.
 */
public String toString()
{
  return getName();
}

/**
 * Sets the crossover operator to be used by the client.
 * @param crossover the crossover opeartor.
 */
public void setCrossover(Crossover crossover)
{
  this.crossover = crossover;
}

/**
 * Sets the mutation operator to be used by the client.
 * @param mutation the mutation operator.
 */
public void setMutation(Mutation mutation)
{
  this.mutation = mutation;
}
* Sets the scenario to be solved by the genetic algorithm.
  * @param scenario the scenario.
  */
public void setScenario(Scenario scenario) throws RemoteException {
  this.scenario = scenario;
}

} // EndOfClass ClientEngine

/**
 * Class to make sure that the client thread can be serialized
 * <p>@copyright Copyright (c) 2004</p>
 * <p>@company Ohio University</p>
 * <p>@author Nihar Shah</p>
 * <p>@version 1.0</p>
 * @ Nihar Shah
 * */
class ClientThread extends Thread implements Serializable {
  /**
   * Constructor that creates a thread object using the given Runnable object
   * @param r the Runnable object
   */
  public ClientThread (Runnable r) {
    super(r);
  }
} // EndOfClass ClientThread
package edu.ohiou.ga.interfaces;

/**
 * Interface that defines a chromosome.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;

public interface Chromosome extends Serializable
{

/**
 * Sets the gene for the given index to the given gene.
 * @param g the gene to be set.
 * @param index the index of the gene to be set.
 * @throws Exception if the index is greater than the current chromosome size.
 */
public void setGene(Gene g, int index) throws Exception;

/**
 * Returns the gene at the given index.
 * @param index the index of the gene.
 * @return the gene in the chromosome.
 * @throws Exception if the index is greater than the current chromosome size.
 */
public Gene getGene(int index) throws Exception;

/**
 * Removes the gene in the given position from the chromosome and returns it.
 * @param index the position of the gene to be removed.
 * @return The removed gene.
 * @throws Exception If the index is greater than the current chromosome length.
 */
public Gene removeGene(int index) throws Exception;

/**
 * Get the fitness value of the chromosome
 * @return the fitness value
 */
public double getFitness();

/**
 * Set the fitness value of the chromosome to the given value
 * @param fitness the fitness value
 */
public void setFitness(double fitness);
/**
 * Return the length of the chromosome.
 * @return the length of the chromosome.
 */

public int getLength();

}//EndOfClass Chromosome
package edu.ohiou.ga.interfaces;

/**
 * This interface provides APIs for the client side of the genetic
 * algorithm.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Client extends Remote {
    /**
     * Allows the server to remotely start the client operation.
     * @return true if the client is started successfully.
     * @throws RemoteException
     */
    public boolean startClientOperation() throws RemoteException;

    /**
     * Allows the server to remotely stop the client operation.
     * @return true if the client is stopped successfully.
     * @throws RemoteException
     */
    public boolean stopClientOperation() throws RemoteException;

    /**
     * Allows the server to remotely shutdown the client.
     * @throws RemoteException
     */
    public void shutDownClient() throws RemoteException;

    /**
     * Provides a string representation of the client. This contains the
     * host address of the client and is of the form
     * &lt;code&gt;/xxx.xxx.xxx/CLIENT &lt;/code&gt;
     * @return the string representation.
     * @throws RemoteException if an error occurs.
     */
    public String getName() throws RemoteException;

    /**
     * Clears the buffer maintained by the client. This API is used at
     * the end of every generation to make sure that the client maintains
     * the chromosomes of the current generation only.
     */
    public void clearBuffer() throws RemoteException;

    /**
     * Sets the scenario to be solved by the genetic algorithm.
     * @param scenario the scenario.
     * @throws RemoteException if scenario can not be set.
     */
}
public void setScenario(Scenario scenario) throws RemoteException;

} // EndOfClass Client
package edu.ohiou.ga.interfaces;

/**
 * Interface that provides APIs to be implemented by crossover
 * operators. These APIs provide access to various crossover parameters.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

public interface Crossover
{

/**
 * Perform the crossover operation on the given chromosomes and
 * return the offspring chromosome.
 * @param ch1 The first parent chromosome.
 * @param ch2 The second parent chromosome.
 * @return the resultant chromosome.
 * @throws Exception if crossover cannot be performed.
 */
public Chromosome doCrossover(Chromosome ch1, Chromosome ch2)
        throws Exception;

/**
 * Returns the probability of crossover occurring on a chromosome.
 * @return the crossover rate.
 */
public double getCrossoverRate();

/**
 * Sets the probability of crossover occurring on a chromosome.
 * @param rate the crossover rate.
 */
public void setCrossoverRate(double rate);

}//EndOfClass Crossover
package edu.ohiou.ga.interfaces;

/**
 * Interface that defines a gene.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

public interface Gene {
    /**
     * Sets the value of the gene to the given value.
     * @param value the value of the gene.
     */
    public void setValue(int value);

    /**
     * Return the value of the gene.
     * @return the value of the gene.
     */
    public int getValue();
}

} // EndOfClass Gene
package edu.ohiou.ga.interfaces;

/**
 * Interface that defines a generation.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

public interface Generation {

    /**
     * Set the array of chromosomes to form the new population
     * @param chromosomes The array of chromosomes.
     */
    public void setChromosomes(Chromosome[] chromosomes);

    /**
     * Sets the chromosome at the specified index in the generation to
     * the given chromosome.
     * @param ch the chromosome to be set.
     * @param index the index of the chromosome to be modified in
     * the generation.
     * @throws Exception if the index is greater than the current
     * chromosome size.
     */
    public void setChromosome(Chromosome ch, int index);

    /**
     * Returns the size of the population contained in the generation
     * @return the population size
     */
    public int getPopulationSize();

    /**
     * Returns the chromosome from the population with the given index.
     * @param chNo the index of the chromosome
     * @return the chromosome
     */
    public Chromosome getChromosome(int chNo);

} // EndOfClass Generation
package edu.ohiou.ga.interfaces;

/**
 * Interface that provides APIs to be implemented by mutation operators.
 * These APIs provide access to various mutation parameters.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

public interface Mutation {

/**
 * Perform the mutation operation on the given chromosome and return
 * the mutated chromosome.
 * @param ch The chromosome to be mutated.
 * @return the resultant chromosome.
 * @throws Exception if mutation cannot be performed.
 */
    public Chromosome doMutation(Chromosome ch) throws Exception;

/**
 * Returns the probability of mutation occurring on a chromosome.
 * @return the mutation rate.
 */
    public double getMutationRate();

/**
 * Returns the maximum number of mutations allowed for each chromosome.
 * @return the maximum number of mutations.
 */
    public int getNumberOfMutations();

/**
 * Set the probability of mutation occurring on a chromosome.
 * @param rate the mutation rate.
 */
    public void setMutationRate(double rate);

/**
 * Set the maximum number of mutations allowed per chromosome.
 * @param num the maximum number of mutations.
 */
    public void setNumberOfMutations(int num);
}

//EndOfClass Mutation
package edu.ohiou.ga.interfaces;

/**
 * Interface that provides APIs to write the output of the genetic
 * algorithm.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

public interface Output
{
    /**
     * Writes the output of the genetic algorithm to a desired output
     * module.
     * @param sc The scenario being solved by the genetic algorithm.
     * @param ch The resultant chromosome of the genetic algorithm.
     */
    public void writeOutput(Scenario sc, Chromosome ch);
}
//EndOfClass Output
package edu.ohiou.ga.interfaces;

/**
 * Interface that provides the APIs to be implemented by sub classes in
 * order to describe a problem to be solved by the genetic algorithm.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;
import java.util.Comparator;

public interface Scenario extends Serializable
{

    /**
     * Create the chromosome data structure based on the given scenario.
     * @return the created chromosome.
     */
    public Chromosome createChromosome();

    /**
     * Evaluate the fitness function of the chromosome based on the
     * scenario.
     * @param ch the chromosome to be evaluated.
     * @return the fitness function value for the chromosome.
     */
    public int evaluateChromosome(Chromosome ch);

    /**
     * The comparator that will be used to compare chromosomes for this
     * scenario.
     * @return the comparator used for comparing chromosomes.
     */
    public Comparator getComparator();

    /**
     * Create the initial population of random chromosomes based on the
     * information given in the scenario.
     * @return the initial generation for the genetic algorithm.
     */
    public Generation createInitialPopulation();

} // EndOfClass Scenario
package edu.ohiou.ga.interfaces;

/**
 * This interface provides APIs for the server side of the distributed
 * genetic algorithm.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Server extends Remote
{
  /**
   * Status of the server when the server is idle and not running
   * the genetic algorithm cycle. In this condition, the server will
   * accept connections.
   */
  public static final int SERVER_IDLE = 0;

  /**
   * Status of the server when the server is running the genetic
   * algorithm cycle and the clients are requesting chromosomes for
   * processing. The server can accept new clients in this mode.
   */
  public static final int SERVER_RUNNING = 1;

  /**
   * Status of the server when the server is running the genetic
   * algorithm cycle and the server's processed buffer is full and is
   * being analyzed to create a chromosome pool for the next generation.
   * The server can accept new clients in this mode.
   */
  public static final int SERVER_PROCESSING = 2;

  /**
   * Allows the remote client to request an unprocessed chromosome from
   * the server's chromosome buffer.
   * @return the unprocessed chromosome from the server's buffer.
   * @throws RemoteException if the chromosome pair cannot be returned.
   */
  public Chromosome[] getChromosomePair() throws RemoteException;

  /**
   * Allows the remote client to put a processed chromosome in the
   * server's result buffer.
   * @param ch the processed chromosome from the client.
   * @return true if the chromosome is successfully returned to the
   * server.
   * @throws RemoteException if the chromosome cannot be inserted
   */
}
public boolean addChromosome(Chromosome ch) throws RemoteException;

/**
 * Allows the remote client to register itself with the server so that
 * the server can notify the client of its operation status.
 * @param client the remote client that is connected to the server.
 * @return true if the client is successfully registered.
 * @throws RemoteException
 */
public boolean registerClient(Client client) throws RemoteException;

/**
 * Allows the remote client to remove itself from the list of active
 * clients held by the server. This method is to be called by the
 * client only when shutting down.
 * @param client the remote client that is to be removed.
 * @return true if the client is successfully removed.
 * @throws RemoteException if a client cannot be removed
 */
public boolean removeClient(Client client) throws RemoteException;

/**
 * Returns the status of the server. It can be one of the following:
 * <ul>
 * <li><b>SERVER_IDLE</b> - if the server is idle and accepting client
 * connections</li>
 * <li><b>SERVER_RUNNING</b> - if the server is accepting client
 * requests for chromosomes</li>
 * <li><b>SERVER_PROCESSING</b> - if the server is analyzing the
 * generation</li>
 * @return the server status
 * @throws RemoteException if the server status cannot be determined
 */
public int serverStatus() throws RemoteException;

} // End Of Class Server
package edu.ohiou.ga.jobshop;

/**
 * Data structure that represents a chromosome in a generation.
 * Stores gene information and provides APIs to access and modify genes.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import edu.ohiou.ga.interfaces.Chromosome;
import edu.ohiou.ga.interfaces.Gene;

public class JSPChromosome implements Chromosome, Cloneable {
    private Gene[] genes;
    private double fitness = 0;

    /**
     * Default constructor that creates a chromosome from the given array of genes.
     * @param genes the genes that constitute this chromosome.
     */
    public JSPChromosome(Gene[] gene) {
        genes = new JSPGene[gene.length];
        for (int i = 0; i < gene.length; i++) {
            genes[i] = new JSPGene(gene[i].getValue());
        }
    }

    /**
     * Sets the gene for the given index to the given gene.
     * @param g the gene to be set.
     * @param index the index of the gene to be set.
     * @throws Exception if the index is greater than the current chromosome length.
     */
    public void setGene(Gene g, int index) throws Exception {
        if (index >= genes.length) {
            throw new Exception("Index is greater than size of current chromosome."
                + "
            "Cannot set gene.");
        }
        genes[index] = new JSPGene(g.getValue());
    }

    /**
     * Removes the gene in the given position from the chromosome and returns it.
     */
}
public Gene removeGene(int index) throws Exception
{
    if (index >= genes.length)
    {
        throw new Exception(
            "Index is greater than size of current chromosome." + '
' +
            "Cannot set gene.");
    }
    JSPGene[] newGenes = new JSPGene[genes.length];
    System.arraycopy(genes, 0, newGenes, 0, genes.length);
    genes = new Gene[newGenes.length - 1];
    System.arraycopy(newGenes, 0, genes, 0, index);
    System.arraycopy(
        newGenes, index + 1, genes, index, genes.length - index);
    return newGenes[index];
}

public Gene getGene(int index) throws Exception
{
    if (index >= genes.length)
    {
        throw new Exception(
            "Index is greater than size of current chromosome." + '
' +
            "Cannot return gene.");
    }
    return genes[index];
}

public Object clone()
{
    return new JSPChromosome((Gene[])genes.clone());
}

public int getLength()
{
    return genes.length;
}
/**
 * Returns the size of the chromosome.
 * @return the size of the chromosome.
 */
public int getChromosomeSize()
{
    return genes.length;
}

/**
 * Returns the string representation of the chromosome.
 * @return the string representation.
 */
public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append("Chromosome" + "");
    for (int i = 0; i < genes.length; i++)
    {
        sb.append(genes[i] + " ");
    }
    return sb.toString();
}

/**
 * Returns the fitness value of the chromosome.
 * @return the fitness value.
 */
public double getFitness()
{
    return fitness;
}

/**
 * Sets the fitness value of the chromosome.
 * @param fitness the fitness value to be set.
 */
public void setFitness(double fitness)
{
    this.fitness = fitness;
}
}//EndOfClass JSPChromosome
package edu.ohiou.ga.jobshop;

/**
 * Simple comparator that is used by the genetic algorithm to sort out
 * chromosomes based on their fitness function values.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */
import java.io.Serializable;
import java.util.Comparator;
import edu.ohiou.ga.interfaces.Chromosome;

public class JSPChromosomeComparator implements Comparator, Serializable {
    /**
     * Default constructor that initializes the comparator.
     */
    public JSPChromosomeComparator()
    {
        //do nothing
    }

    /**
     * Overrides the <code>compare()</code> method in
     * <code>Comparator</code> to compare two hromosomes based on their
     * fitness function values.
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return the difference between the fitness functions of the two
     * chromosomes.
     */
    public int compare(Object o1, Object o2)
    {
        Chromosome ch1 = (Chromosome) o1;
        Chromosome ch2 = (Chromosome) o2;
        return Math.round( (float) (ch1.getFitness() - ch2.getFitness()));
    }

    /**
     * Overrides the <code>equals()</code> method in
     * <code>Comparator</code> to determine whether the given object is
     * the same as this comparator object.
     * @param obj the object to be compared to this comparator.
     * @return true if the object is same as the comparator, false
     * otherwise.
     */
    public boolean equals(Object obj)
    {
        if (obj instanceof JSPChromosomeComparator)
        {
            // do nothing
        }
        return false;
    }
}
return true;
}
else{
    return false;
}

 }//EndOfClass ChromosomeComparator
package edu.ohiou.ga.jobshop;

/**
 * Data structure that represents the genes in a chromosome of a genetic
 * algorithm.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;

import edu.ohiou.ga.interfaces.Gene;

public class JSPGene implements Serializable, Gene, Cloneable
{
    private int value;

    /**
     * Default constructor that creates a gene from the given value.
     * @param value the value of the gene.
     */
    public JSPGene(int value)
    {
        this.value = value;
    }

    /**
     * Returns the current value of the gene.
     * @return the current gene value.
     */
    public int getValue()
    {
        return value;
    }

    /**
     * Sets the value of the gene to the given value.
     * @param value the value of the gene.
     */
    public void setValue(int value)
    {
        this.value = value;
    }

    /**
     * Returns a string representation of the gene's value.
     * @return the string representation.
     */
    public String toString()
    {
        return Integer.toString(value);
    }
}
/**
 * Clone this object
 * @return a clone of this object
 */
public Object clone()
{
    return new JSPGene(value);
}

}//EndOfClass JSPGene
package edu.ohiou.ga.jobshop;

/**
 * Data structure that represents a generation of chromosomes in the genetic algorithm. Consists of an array of chromosomes and various APIs to access these chromosomes.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;
import edu.ohiou.ga.interfaces.Chromosome;
import edu.ohiou.ga.interfaces.Generation;

public class JSPGeneration implements Serializable, Generation {
    private Chromosome[] chromosomes;

    /**
     * Default constructor that initializes a generation from the given chromosomes.
     * @param chromosomes the chromosomes that form the generation.
     */
    public JSPGeneration(Chromosome[] chromosomes) {
        setChromosomes(chromosomes);
    }

    /**
     * Set the array of chromosomes to form the new population
     * @param chromosomes The array of chromosomes.
     */
    public void setChromosomes(Chromosome[] chromosomes) {
        this.chromosomes = new Chromosome[chromosomes.length];
        for (int i = 0; i < chromosomes.length; i++) {
            this.chromosomes[i] = chromosomes[i];
        }
    }

    /**
     * Sets the chromosome at the specified index in the generation to the given chromosome.
     * @param ch the chromosome to be set.
     * @param index the index of the chromosome to be modified in the generation.
     * @throws Exception if the index is greater than the current chromosome size.
     */
    public void setChromosome(Chromosome ch, int index)
```java
if (index >= chromosomes.length)
    throw new ArrayIndexOutOfBoundsException("Index is greater than size of current generation."
        + '\n' + "Cannot set chromosome.");

chromosomes[index] = ch;

/**
 * Returns the chromosome at the given index.
 * @param index the index of the chromosome.
 * @return the chromosome.
 * @throws Exception if the index is greater than the current
 *        generation size.
 */
public Chromosome getChromosome(int index)
{
    if (index >= chromosomes.length)
        throw new ArrayIndexOutOfBoundsException("Index is greater than size of current generation."
            + '\n' + "Cannot set chromosome.");

    return chromosomes[index];
}

/**
 * The size of the generation.
 * @return the size.
 */
public int getPopulationSize()
{
    return chromosomes.length;
}
```
package edu.ohiou.ga.jobshop;

/**<n*
* Scenario Parser that parses the input xml file and populates the data
* structure for the scenario.
* @copyright Copyright (c) 2004
* @company Ohio University
* @author Nihar Shah
* @version 1.0
*/
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import edu.ohiou.ga.GaParameters;
import edu.ohiou.ga.interfaces.Scenario;
public class JSPHandler extends DefaultHandler{
    private Scenario readScenario;
    private int jobs = 0;
    private int macs = 0;
    private int jobNo = 0;
    private int opNo = 0;
    private int macNo = 0;
    private int time = 0;
    private int dd = 0;
    private String scenarioID;
    private int optSol = 0;
    private GaParameters gaParams;
    public JSPHandler(GaParameters params){
        gaParams = params;
    }
    /**
     * Receive notification of the beginning of a document.
     * @exception org.xml.sax.SAXException Any SAX exception, possibly
     * wrapping another exception.
     */
    public void startDocument() throws SAXException{
        //Do nothing
    }
    /**
     * Receive notification of the end of a document.
     * @exception org.xml.sax.SAXException Any SAX exception, possibly
     * wrapping another exception.
     */
    public void endDocument() throws SAXException{
        //DO nothing
    }
/**
 * Receive notification of the beginning of an element.
 * The Parser will invoke this method at the beginning of every
 * element in the XML document; there will be a corresponding
 * (endElement) event for every startElement event
 * (even when the element is empty). All of the element's content will be
 * reported, in order, before the corresponding endElement
 * event.</p>
 * @param namespaceURI The Namespace URI, or the empty string if the
 * element has no Namespace URI or if Namespace processing is not
 * being performed.
 * @param sName The local name (without prefix), or the
 * empty string if Namespace processing is not being performed.
 * @param qName The qualified name (with prefix), or the
 * empty string if qualified names are not available.
 * @param attrs The attributes attached to the element. If
 * there are no attributes, it shall be an empty Attributes object.
 * @exception org.xml.sax.SAXException Any SAX exception, possibly
 * wrapping another exception.
 */
public void startElement(String namespaceURI, String sName,
        String qName, Attributes attrs) throws SAXException
{
    String eName = sName;
    if ("".equals(eName))
    {
        eName = qName;
    }
    if (eName.equals("Scenario") && (attrs != null))
    {
        for (int i = 0; i < attrs.getLength(); i++)
        {
            String aName = attrs.getLocalName(i);
            if ("".equals(aName))
            {
                aName = attrs.getQName(i);
            }
            String val = attrs.getValue(i).trim();
            if (aName.equals("ScenarioID"))
            {
                scenarioID = new String(val);
            }
            if (aName.equals("OptimalSolution"))
            {
                optSol = Integer.parseInt(val);
            }
            if (aName.equals("Jobs"))
            {
                jobs = Integer.parseInt(val);
            }
            if (aName.equals("Machines"))
            {
                machines = Integer.parseInt(val);
            }
        }
    }
}
readScenario = new JSPScenario(scenarioID, gaParams, jobs, macs, optSol);

if (eName.equals("Job") && (attrs != null))
{
    for (int i = 0; i < attrs.getLength(); i++)
    {
        String aName = attrs.getLocalName(i);
        if ("".equals(aName))
            aName = attrs.getQName(i);

        String val = attrs.getValue(i).trim();
        if (aName.equals("JobID"))
        {
            jobNo = Integer.parseInt(val) - 1;
        }
        if (aName.equals("DueDate"))
        {
            dd = Integer.parseInt(val);
        }
    }
}

if (eName.equals("Operation") && (attrs != null))
{
    for (int i = 0; i < attrs.getLength(); i++)
    {
        String aName = attrs.getLocalName(i);
        if ("".equals(aName))
            aName = attrs.getQName(i);

        String val = attrs.getValue(i).trim();
        if (aName.equals("OperationID"))
        {
            opNo = Integer.parseInt(val) - 1;
        }
        if (aName.equals("MachineID"))
        {
            macNo = Integer.parseInt(val) - 1;
        }
        if (aName.equals("Time"))
        {
            time = Integer.parseInt(val);
        }
    }
}
/**
 * Receive notification of the end of an element.
 * <p>The SAX parser will invoke this method at the end of every
 * element in the XML document; there will be a corresponding
 * (@link #startElement startElement) event for every endElement
 * event (even when the element is empty).</p>
 * @param namespaceURI The Namespace URI, or the empty string if the
 * element has no Namespace URI or if Namespace processing is not
 * being performed.
 * @param sName The local name (without prefix), or the
 * empty string if Namespace processing is not being performed.
 * @param qName The qualified XML 1.0 name (with prefix), or the
 * empty string if qualified names are not available.
 * @exception org.xml.sax.SAXException Any SAX exception, possibly
 * wrapping another exception.
 */
public void endElement(String namespaceURI, String sName,
String qName) throws SAXException
{
String eName = sName;
if ("".equals(eName))
{
eName = qName;
}
if (eName.equals("Operation"))
{
    JSPJob j = null;
    JSPMachine m = null;
    try
    {
        j = ( (JSPScenario) readScenario).getJob(jobNo);
        m = ( (JSPScenario) readScenario).getMachine(macNo);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        System.exit(1);
    }
    j.addOperation(opNo, m, time);
}
/**
 * Returns the scenario created by parsing the XML file.
 * @return The scenario
 */
public Scenario getScenario()
{
    return readScenario;
}
package edu.ohiou.ga.jobshop;

/**
 * Data structure that defines a job in a job shop environment.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */
import java.util.ArrayList;
import java.util.List;
public class JSPJob implements java.io.Serializable {
    /**
     * The list containing the operations to be performed to complete this
     * job.
     */
    private List operationList = new ArrayList();

    /**
     * The job number of the job that is being performed.
     */
    private int jobNumber;

    /**
     * Creates a job with the specified number of operations
     * @param jobNo The job number of the job
     */
    public JSPJob(int jobNo) {
        jobNumber = jobNo;
    }

    /**
     * Returns the number of machines required by the job
     * @return The number of machines
     */
    public int getNumberOfOperations() {
        return operationList.size();
    }

    /**
     * Returns the operation specified by opNo for the job
     * @param opNo The operation number
     * @return The Operation
     */
    public JSPOperation getOperation(int opNo) {
        return (JSPOperation) operationList.get(opNo);
    }

    /**
     * Set the operation specified by opNo to be performed on the
     * @param opNo The operation number
     */
    public void setOperation(int opNo, JSPOperation operation) {
        operationList.set(opNo, operation);
    }
}

// End of JSPJob class
public void addOperation(int opNo, JSPMachine m, int time) {
    operationList.add(new JSPOperation(opNo, m, time));
}

/**
 * Returns the job number of the job
 * @return The job number
 */
public int getJobNumber() {
    return jobNumber;
}

/**
 * Sets the job number of the job
 * @param jobNumber The job number to be set
 */
public void setJobNumber(int jobNumber) {
    this.jobNumber = jobNumber;
}

}//EndOfClass JSPJob
package edu.ohiou.ga.jobshop;

/**
 * Data structure that represents a machine in a job shop environment.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;

public class JSPMachine implements Serializable
{
    private int machineNo;

    /**
     * Default constructor which initializes a machine with a machine number
     * @param machNo The machine number of the machine
     */
    public JSPMachine(int machNo)
    {
        machineNo = machNo;
    }

    /**
     * Returns the machine number of the machine
     * @return The machine number
     */
    public int getMachineNo()
    {
        return machineNo;
    }

    /**
     * Sets the machine number of the machine
     * @param machNo The machine number to be set
     */
    public void setMachineNo(int machNo)
    {
        machineNo = machNo;
    }
}

}//EndOfClass JSPMachine
package edu.ohiou.ga.jobshop;

/**
 * Data structure that represents an operation in a job shop problem.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;

public class JSPOperation implements Serializable {

    private int operationNo;
    private int machiningTime;
    private JSPMachine mfgMachine;

    /**
     * Constructor to create an operation which has machine <b>mach</b> and machining time <b>time</b>
     * @param mach The <code>Machine</code> on which the operation is to be performed
     * @param opNo The operation number of the operation
     * @param time The time taken to complete the operation
     */
    public JSPOperation(int opNo, JSPMachine mach, int time) {
        mfgMachine = mach;
        machiningTime = time;
        operationNo = opNo;
    }

    /**
     * Returns the machine on which the operation is to be performed
     * @return The machine
     */
    public JSPMachine getMachine() {
        return mfgMachine;
    }

    /**
     * Returns the machining time required by the operation
     * @return The machining time
     */
    public int getMachingTime() {
        return machiningTime;
    }

    /**
     * Returns the operation number of the operation
     * @return The operation number
     */
}
```java
public int getOperationNumber()
{
    return operationNo;
}

/**
 * Sets the machine on which the operation is to be performed
 * @param m The machine to be set
 */
public void setMachine(JSPMachine m)
{
    mfgMachine = m;
}

/**
 * Sets the machining time required by the operation
 * @param time The machining time of the operation
 */
public void setMachiningTime(int time)
{
    machiningTime = time;
}

/**
 * Sets the operation number of the operation
 * @param opNo The operation number to be set
 */
public void setOperationNumber(int opNo)
{
    operationNo = opNo;
}

} // EndOfClass JSPOperation
```
package edu.ohiou.ga.jobshop;

/**
 * This class implements order crossover as proposed by Davis.
 * A substring is selected from one parent at random and copied into
 * corresponding positions in the offspring chromosome. The genes of this
 * sequence are deleted from the second parent and then the second parent's
 * genes are copied into the empty positions in the offspring.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */
import edu.ohiou.ga.interfaces.*;

public class JSPOrderCrossover implements Crossover {

    private double crossoverRate = 0.0;
    /**
     * Default constructor
     */
    public JSPOrderCrossover() {
        //do nothing
    }

    /**
     * Constructor to create crossover operator with the desired crossover
     * probability
     * @param crossOverProb The crossover probability of the operator
     */
    public JSPOrderCrossover(double crossOverProb) {
        crossoverRate = crossOverProb;
    }

    /**
     * Returns the probability of crossover occurring on a chromosome.
     * @return the crossover rate.
     */
    public double getCrossoverRate() {
        return crossoverRate;
    }

    /**
     * Sets the probability of crossover occurring on a chromosome.
     * @param rate the crossover rate.
     */
    public void setCrossoverRate(double rate) {
        crossoverRate = rate;
    }
}
/**
 * Performs the cross over operation between chromosomes c1 and c2
 * if the random number generated is less than the crossover
 * probability
 * @param c1 The first parent chromosome
 * @param c2 The second parent chromosome
 * @return The offspring chromosome if cross over occurs, else c1
 * @throws Exception if crossover cannot be performed
 */

public Chromosome doCrossover(Chromosome c1, Chromosome c2)
throws Exception
{
    Gene[] genes = new Gene[c1.getLength()];
    Chromosome cx2 = (Chromosome)((JSPChromosome)c2).clone();

    double x = Math.random();
    int pos1 = 0;
    int pos2 = 0;
    pos1 = (int) (Math.random() * 1000) % c1.getLength();
    pos2 = (int) (Math.random() * 1000) % c1.getLength();

    while (pos1 > pos2)
    {
        pos2 = (int) (Math.random() * 1000) % c1.getLength();
    }

    if (x <= crossoverRate)
    {
        for (int i = pos1; i < pos2; i++)
        {
            genes[i] = new JSPGene(c1.getGene(i).getValue());

            boolean remove = true;
            int count = cx2.getLength() - 1;
            while (remove && (count >= 0))
            {
                if (cx2.getGene(count).getValue() == c1.getGene(i).getValue())
                {
                    cx2.removeGene(count);
                    remove = false;
                }
                else
                {
                    count--;
                }
            }
        }
    }
    int count = 0;

    for (int i = 0; i < c1.getLength(); i++)
    {
        if ( (i < pos1) || (i >= pos2))
        {
            genes[i] = new JSPGene(cx2.getGene(count).getValue());
            count++;
        }
    }

    return new JSPChromosome(genes);
if (c2 == cl)
{
    return c2;
}
else
{
    return cl;
}

}//EndOfClass JSPOrderCrossover
package edu.ohiou.ga.jobshop;

/**
 * This class implements pair exchange mutation. Two genes are selected
 * at random and their positions are swapped to produce resultant chromosome.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */
import edu.ohiou.ga.interfaces.Chromosome;
import edu.ohiou.ga.interfaces.Mutation;

public class JSPOrderMutation implements Mutation {
    private double mutationRate;
    private int numberOfMutations;
    /**
     * Default constructor
     */
    public JSPOrderMutation()
    {
        //do nothing
    }
    /**
     * Constructor to initialize mutation operator with specified
     * mutation rate and specified number of mutations.
     * @param mutationProb The mutation probability of the GA.
     * @param numOfMutations The number of mutations to be performed
     * for each chromosome
     */
    public JSPOrderMutation(double mutationProb, int numOfMutations)
    {
        mutationRate = mutationProb;
        numberOfMutations = numOfMutations;
    }
    /**
     * Perform the mutation operation on the given chromosome and return
     * the mutated chromosome.
     * @param ch The chromosome to be mutated.
     * @return the resultant chromosome.
     * @throws Exception if mutation cannot be performed.
     */
    public Chromosome doMutation(Chromosome ch) throws Exception
    {
        for (int i = 0; i < numberOfMutations; i++)
        {
            double x = Math.random();
            if (x <= mutationRate)
            {
int pos1 = (int) (Math.random() * 1000) % ch.getLength();
int pos2 = (int) (Math.random() * 1000) % ch.getLength();
int g1 = ch.getGene(pos1).getValue();
int g2 = ch.getGene(pos2).getValue();
ch.getGene(pos1).setValue(g2);
ch.getGene(pos2).setValue(g1);
return ch;
}

/**
 * Returns the probability of mutation occurring on a chromosome.
 * @return the mutation rate.
 */
public double getMutationRate()
{
    return mutationRate;
}

/**
 * Returns the maximum number of mutations allowed per chromosome.
 * @return the maximum number of mutations.
 */
public int getNumberOfMutations()
{
    return numberOfMutations;
}

/**
 * Set the probability of mutation occurring on a chromosome.
 * @param rate the mutation rate.
 */
public void setMutationRate(double rate)
{
    mutationRate = rate;
}

/**
 * Set the maximum number of mutations allowed per chromosome.
 * @param num the maximum number of mutations.
 */
public void setNumberOfMutations(int num)
{
    numberOfMutations = num;
}

}//EndOfClass JSPOrderMutation
package edu.ohiou.ga.jobshop;

/**
 * This class defines the job shop problem data structure and implements the
 * <code>Scenario</code> interface. The input file is read using the XML
 * parser and appropriate data structures are populated. Fitness function is
 * also calculated using this class.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.File;
import java.io.IOException;
import java.util.Comparator;
import javax.xml.parsers *
import org.xml.sax.SAXException;
import edu.ohiou.ga.GaParameters;
import edu.ohiou.ga.interfaces *

public class JSPScenario implements Scenario
{

/**
 * A unique identifier for the scenario
 */
private String scenarioID;

/**
 * The number of jobs in the scenario
 */
private int noOfJobs;

/**
 * The number of machines in the scenario
 */
private int noOfMachines;

/**
 * The optimal solution for the scenario
 */
private int optimalSol;

/**
 * The jobs to be performed in the scenario
 */
private JSPJob[] jobs;

/**
 * The genetic algorithm parameters
 */
private GaParameters gaParams;
/**
 * The machines used in performing jobs in the scenario
 */
private JSPMachine[] machines;

/**
 * Handle to the comparator used for sorting the chromosomes
 */
private JSPChromosomeComparator comparator;

/**
 * Creates the job shop scenario from the given file.
 * @param scenarioFile the file containing information on the problem.
 * @param params the genetic algorithm parameters.
 * @return The created data structure.
 * @throws Exception If an error occurs in reading the file or if the file
 * format is incorrect.
 */
public static Scenario createScenario(File scenarioFile,
GaParameters params) throws Exception {
    JSPHandler handler = new JSPHandler(params);
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser saxParser = null;
    try {
        saxParser = factory.newSAXParser();
        saxParser.parse(scenarioFile, handler);
    }
    catch (SAXException ex) {
        throw new Exception("Error reading XML File");
    }
    catch (ParserConfigurationException ex) {
        throw new Exception("Cannot create parser");
    }
    catch (IOException ex) {
        throw new Exception("Cannot read XML File");
    }
    return handler.getScenario();
}

/**
 * Constructor that creates a scenario from the given id, number of jobs
 * and machines and the optimal solution
 * @param scenarioID The unique identifier for the scenario
 * @param params The genetic algorithm parameters
 * @param noOfJobs The number of jobs in the scenario
 * @param noOfMachines The number of machines in the scenario
 * @param optimalSol The optimal solution of the scenario
 */
public JSPScenario(String scenarioID, GaParameters params, int noOfJobs,
    int noOfMachines, int optimalSol) {
    this.scenarioID = scenarioID;
    gaParams = params;
this.noOfJobs = noOfJobs;
jobs = new JSPJob[noOfJobs];
for (int i = 0; i < jobs.length; i++)
{
    jobs[i] = new JSPJob(i);
}
this.noOfMachines = noOfMachines;
machines = new JSPMachine[noOfMachines];
for (int i = 0; i < machines.length; i++)
{
    machines[i] = new JSPMachine(i);
}
this.optimalSol = optimalSol;

/**
 * Create the chromosome data structure based on the given scenario.
 * @return the created chromosome.
 */
public Chromosome createChromosome()
{
    //determine the length of the chromosome
    int len = 0;
    for (int i = 0; i < noOfJobs; i++)
    {
        len += jobs[i].getNumberOfOperations();
    }
    //Create the array of genes and populate it
    Gene[] genes = new JSPGene[len];
    int count = 0;
    int[] jobCount = new int[noOfJobs + 1];
    int x = 0;
    while (count < len)
    {
        double randNo = Math.random();
        int geneValue = (int) (randNo * 1000) % noOfJobs + 1;
        if (jobCount[geneValue] < jobs[geneValue - 1].getNumberOfOperations())
        {
            genes[x] = new JSPGene(geneValue);
            x++;
            count++;
            jobCount[geneValue]++;
        }
    }
    Chromosome c = new JSPChromosome(genes);
    c.setFitness(Integer.MAX_VALUE);
    return c;
}

/**
 * Create the initial population of random chromosomes based on the
 * information given in the scenario.
 * @return the initial generation for the genetic algorithm.
 */
public Generation createInitialPopulation()
{
    Chromosome[] chs = new JSPChromosome[gaParams.populationSize()];
    for (int i = 0; i < gaParams.populationSize(); i++)
/**
 * Evaluate the fitness function of the chromosome based on the scenario.
 * @param ch the chromosome to be evaluated.
 * @return the fitness function value for the chromosome.
 */
public int evaluateChromosome(Chromosome ch)
{
    int[] machineTime = new int[machines.length];
    int[] jobTime = new int[jobs.length];
    int[] jobCount = new int[jobs.length];
    int value = 0;
    int jobNo = 0;
    int macNo = 0;
    int operNo = 0;
    int time = 0;
    JSPOperation op;
    try
    {
        for (int i = 0; i < ch.getLength(); i++)
        {
            jobNo = ch.getGene(i).getValue() - 1;
            operNo = jobCount[jobNo];
            op = jobs[jobNo].getOperation(operNo);
            macNo = op.getMachine().getMachineNo();
            time = op.getMachiningTime();
            if (jobTime[jobNo] > machineTime[macNo])
            {
                machineTime[macNo] = jobTime[jobNo];
            }
            else
            {
                jobTime[jobNo] = machineTime[macNo];
            }
            jobTime[jobNo] += time;
            machineTime[macNo] += time;
            jobCount[jobNo]++;
        }
        value = getMax(machineTime);
        ch.setFitness(value);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
    return value;
}

/**
 * Finds the maximum value of any index in the array.
 * @param array The array to be searched.
 * @return The maximum value.
 */
private int getMax(int[] array)
```java
{ int temp = 0;
for (int i = 0; i < array.length; i++)
{
    if (array[i] > temp)
    {
        temp = array[i];
    }
}
return temp;
}

/**
* The comparator that will be used to compare chromosomes for this
* scenario.
* @return the comparator used for comparing chromosomes.
*/
public Comparator getComparator()
{
    if (comparator == null)
    {
        comparator = new JSPChromosomeComparator();
    }
    return comparator;
}

/**
* Returns the job associated with the given job no.
* @param jobNo the job number of the job.
* @return the job
*/
public JSPJob getJob(int jobNo)
{
    return jobs[jobNo];
}

/**
* Returns the machine associated with the given machine no.
* @param macNo the machine number of the machine.
* @return the machine
*/
public JSPMachine getMachine(int macNo)
{
    return machines[macNo];
}
} // EndOfClass JSPScenario
```
package edu.ohiou.ga.server;

/**
 * Main class that controls the genetic algorithm server.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.File;
import edu.ohiou.ga.GaParameters;
import edu.ohiou.ga.interfaces.Scenario;
import edu.ohiou.ga.jobshop.JSPScenario;

public class ServerControl
{
    public static int noOfClients = 0;

    public static void main(String[] args)
    {
        if (args == null || args.length != 12)
        {
            System.out.println("Incorrect parameters specified.");
            System.exit(-1);
        }

        double cProb = 0.0; // crossover probability
        double mProb = 0.0; // mutation probability
        int nGen = 0; // number of generations
        int pSize = 0; // population size
        File inFile = null; // input scenario file
        int i = 0;
        while (i < args.length)
        {
            if (args[i].startsWith("-"))
            {
                char flag = args[i].charAt(1);
                switch (flag)
                {
                    case 'c':
                        // crossover probability
                        cProb = Double.parseDouble(args[++i]);
                        break;
                    case 'm':
                        // mutation probability
                        mProb = Double.parseDouble(args[++i]);
                        break;
                    case 'g':
                        // number of generations
                        nGen = Integer.parseInt(args[++i]);
                        break;
                    default:
                        System.out.println("Invalid flag: "+flag);
                        break;
                }
                i++;
            }
            inFile = new File(args[i]); // input scenario file
            inFile = null;
            i++;
        }
nGen = Integer.parseInt(args[++i]);
break;

case 'p':
    // population size
    pSize = Integer.parseInt(args[++i]);
    break;

case 'f':
    // scenario file
    inFile = new File(args[++i]);
    break;

case 'n':
    // no of clients
    noOfClients = Integer.parseInt(args[++i]);
    break;

default:
    System.out.println("Invalid option: " + flag);
    break;

    }
    i++;
}

GaParameters params = new GaParameters(cProb, mProb, pSize, nGen);
Scenario s = null;
ServerEngine eng = null;
try
{
    s = JSPScenario.createScenario(inFile, params);
    eng = new ServerEngine(s, params);
    eng.startServer();
    eng.execute();
}
catch (Exception ex)
{
    ex.printStackTrace();
}

}//EndOfClass ServerControl
package edu.ohiou.ga.server;

/**
 * Server implementation of the genetic algorithm. The server maintains
 * two buffers, one containing unprocessed chromosomes to be processed by
 * the clients and one containing chromosomes that have been processed by
 * the clients and returned to the server. The server also maintains a
 * list of clients connected to it and keeps updating this list
 * dynamically as clients are added and removed from the server.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

import java.io.Serializable;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
import edu.ohiou.ga.*;
import edu.ohiou.ga.interfaces.*;

public class ServerEngine extends UnicastRemoteObject
        implements Server, Serializable
{
    /**
     * The map containing the clients connected to the server.
     */
    private SortedMap clientMap = new TreeMap();

    /**
     * The list containing unprocessed chromosomes.
     */
    private LinkedList rawList = new LinkedList();

    /**
     * The list containing processed chromosomes.
     */
    private LinkedList processedList = new LinkedList();

    /**
     * The number of generations that the genetic algorithm should run.
     */
    private int totalGenerations = 0;

    /**
     * The number of the current generation that is being executed by the
     * genetic algorithm.
     */
    private int curGeneration = 0;
/**
 * The generation being executed by the genetic algorithm.
 */
private Generation generation;

private int populationSize = 0;

/**
 * The comparator used for sorting chromosomes.
 */
private Comparator comparator;

/**
 * The Server thread that does all the work
 */
private Thread gaServer;

/**
 * The data structure that holds the results generated at the end of
 * every generation
 */
private GaResults results;

/**
 * The scenario information on the problem being solved
 */
private Scenario gaScenario;

/**
 * The parameters of the genetic algorithm
 */
private GaParameters gaParams;

/**
 * The status of the server
 */
private int serverStatus = Server.SERVER_IDLE;

private long startTime;
private long endTime;

/**
 * Default constructor that initializes the server
 * @param s Scenario that is to be solved using the genetic algorithm.
 * @param params The genetic algorithm parameters.
 * @throws RemoteException If the engine cannot be constructed
 */
public ServerEngine(Scenario s, GaParameters params)
    throws RemoteException
{
    super();
    gaScenario = s;
    gaParams = params;
    totalGenerations = params.numberOfGenerations();
    populationSize = params.populationSize();
    gaServer = new ServerThread();
    comparator = s.getComparator();
    results = new GaResults();
}
/**
 * Create the RMI server object and bind it to the machine name.
 * @throws Exception thrown if server cannot be started.
 */
public void startServer() throws Exception
{
    String serverName;
    try {
        serverName = InetAddress.getLocalHost().getHostName();
    }
    catch (UnknownHostException ex) {
        System.out.println("Host name not resolved... "+"Setting default value as \'localhost\'");
        serverName = "localhost";
    }
    serverName = "/" + serverName + "/Server";
    System.out.println("--> " + serverName);
    Naming.rebind(serverName, this);
}

/**
 * Allows the remote client to request an unprocessed chromosome from
 * the server's chromosome buffer.
 * @return the unprocessed chromosome from the server's buffer.
 * @throws RemoteException
 */
public Chromosome[] getChromosomePair() throws RemoteException
{
    synchronized (rawList)
    {
        Chromosome[] ch = new Chromosome[2];
        if (rawList.size() < 2)
        {
            ch[0] = gaScenario.createChromosome();
            ch[1] = gaScenario.createChromosome();
        }
        else
        {
            ch[0] = (Chromosome) rawList.removeFirst();
            ch[1] = (Chromosome) rawList.removeFirst();
        }
        return ch;
    }
}

/**
 * Allows the remote client to put a processed chromosome in the
 * server's result buffer.
 * @param ch the processed chromosome from the client.
 * @return true if the chromosome is successfully returned to the
 * server.
 * @throws RemoteException
 */
public boolean addChromosome(Chromosome ch) throws RemoteException
{
synchronized (processedList) {
    try {
        processedList.add(ch);
    } catch (Exception ex) {
        return false;
    }
}
synchronized (gaServer) {
    if (processedList.size() >= populationSize) {
        serverStatus = Server.SERVER_PROCESSING;
        gaServer.notify();
    }
}
return true;

/**
 * Allows the remote client to register itself with the server so that
 * the server can notify the client of its operation status.
 * @param client the remote client that is connected to the server.
 * @return true if the client is successfully registered.
 * @throws RemoteException
 */
public boolean registerClient(Client client) throws RemoteException {
    System.out.println("ServerEngine: registerClient() "+ client.getName());
    try {
        if (clientMap.get(client.getName()) != null) {
            System.out.println("Reconnecting " + client.getName());
            clientMap.remove(client.getName());
        }
        String clName = client.getName();
        Client c = (Client) Naming.lookup(clName);
        c.setScenario(gaScenario);
        clientMap.put(c.getName(), c);
        System.out.println("Added " + c.getName());
        if (serverStatus == Server.SERVER_RUNNING) {
            // notify the client to start processing
            c.startClientOperation();
            System.out.println("Client " + c.getName() + " started.");
        }
        return true;
    } catch (Exception ex) {
        ex.printStackTrace();
        return false;
    }
}
/**
 * Allows the remote client to remove itself from the list of
 * active clients held by the server. This method is to be called by
 * the client only when shutting down.
 * @param client the remote client that is to be removed.
 * @return true if the client is successfully removed.
 * @throws RemoteException
 */
public boolean removeClient(Client client) throws RemoteException {
    System.out.println("ServerEngine: removeClient()");
    if (clientMap.containsKey(client.toString())) {
        clientMap.remove(client.toString());
        System.out.println("Removed " + client);
        return true;
    } else {
        return false;
    }
}

/**
 * Returns the status of the server. It can be one of the following:
 * <ul>
 * <li><b>SERVER_IDLE</b> - if the server is idle and accepting
 * client connections</li>
 * <li><b>SERVER_RUNNING</b> - if the server is accepting client
 * requests for chromosomes</li>
 * <li><b>SERVER_PROCESSING</b> - if the server is analyzing the
 * generation</li>
 * @return the server status
 * @throws RemoteException
 */
public int serverStatus() throws RemoteException {
    return serverStatus;
}

/**
 * Executes the genetic algorithm
 */
public void execute() {
    System.out.println("ServerEngine: execute()");
    //Step 1
    //Create initial population
    generation = gaScenario.createInitialPopulation();
    int counter = 0;

    //Populate rawList using the initial generation
    rawList.clear();
    int size = generation.getPopulationSize();
    for (int i = 0; i < size; i++) {
    }
rawList.add(generation.getChromosome(i));
rawList.add(generation.getChromosome(size - i - 1));
}
Collections.shuffle(rawList,
    new Random(System.currentTimeMillis()));

//Check whether number of clients is adequate
while (clientMap.size() < ServerControl.noOfClients)
{
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException ex)
    {
    }
}
//Start timing
startTime = System.currentTimeMillis();
System.out.println("Started at "+ startTime);

//Start the ga operation
serverStatus = Server.SERVER_RUNNING;
//notify connected clients
startClients();
gaServer.start();
}

/**
 * Thread that executes the server cycle
 * <p>@copyright Copyright (c) 2004</p>
 * <p>@company Ohio University</p>
 * <p>@author Nihar Shah</p>
 * <p>@version 1.0</p>
 * Nihar Shah
 * */
class ServerThread extends Thread implements Serializable
{
    int genCounter = 0;
    public ServerThread()
    {
        //Create the initial population
    }
    /**
     * Thread that executes the genetic algorithm cycle. Processes
     * chromosomes to select the best chromosomes for the next
     * generation. Puts these selected chromosomes in the buffer for
     * the clients to access and process.
     */
    public void run()
    {
        //When new population is generated stop all clients and
        //process new generation
        while (genCounter < totalGenerations)
        {
            while (processedList.size() < populationSize)
            {
                try
sleep(10);
}
catch (Exception ex)
{
}

//stopClients();
System.out.println("Generation: " + genCounter);
synchronized (processedList)
{
    Chromosome[] array = new Chromosome[processedList.size()];
    processedList = (LinkedList) sortChromosomes(processedList);
    processedList.toArray(array);
    Chromosome bestCh = array[0];
    generation.setChromosomes(array);
    results.storeResults(generation, genCounter);
    rawList = (LinkedList) selectChromosomes(
        calculateProbabilities());
    processedList.clear();
    processedList.add(bestCh);
}
//startClients();
genCounter++;

endTime = System.currentTimeMillis();
long execTime = endTime - startTime;
System.out.println("Operation finished in "
    + (execTime / 1000.0) + " seconds");

shutDownClients();

ResultData best = results.getResults(0);
ResultData worst = results.getResults(0);

for (int i = 0; i < totalGenerations; i++)
{
    ResultData data = results.getResults(i);
    if (data.getBestChromosome().getFitness() < best.getBestChromosome().getFitness())
    {
        best = results.getResults(i);
    }
    if (worst.getWorstChromosome().getFitness() > data.getWorstChromosome().getFitness())
    {
        worst = results.getResults(i);
    }
}

System.out.println("Best: " + best.getGeneration());
System.out.println(best.getBestChromosome());
System.out.println("Fitness: "
    + best.getBestChromosome().getFitness());
System.out.println("-----");
System.out.println("Worst: " + worst.getGeneration());
System.out.println(worst.getWorstChromosome());
System.out.println("Fitness: "
    + worst.getWorstChromosome().getFitness());
+ worst.getWorstChromosome().getFitness());
System.exit(0);
}
}

/**
 * Iterate over the list of clients and notify them to start
 * their processing of the chromosomes. If any client can not be
 * notified, it is removed from the list of active clients
 */
public void startClients()
{
    String[] clientNames = new String[clientMap.size()];
    clientMap.keySet().toArray(clientNames);
    for (int i = 0; i < clientNames.length; i++)
    {
        Client client = (Client) clientMap.get(clientNames[i]);
        try
        {
            client.startClientOperation();
        }
        catch (RemoteException ex)
        {
            clientMap.remove(clientNames[i]);
        }
    }
}

/**
 * Iterate over the list of clients and clear their buffers.
 */
public void clearClients()
{
    String[] clientNames = new String[clientMap.size()];
    clientMap.keySet().toArray(clientNames);
    for (int i = 0; i < clientNames.length; i++)
    {
        Client client = (Client) clientMap.get(clientNames[i]);
        try
        {
            client.clearBuffer();
        }
        catch (RemoteException ex)
        {
            clientMap.remove(clientNames[i]);
        }
    }
}

/**
 * Iterate over the list of clients and notify them to stop
 * their processing of the chromosomes. If any client can not be
 * notified, it is removed from the list of active clients
 */
public void stopClients()
{
    String[] clientNames = new String[clientMap.size()];
    clientMap.keySet().toArray(clientNames);
    for (int i = 0; i < clientNames.length; i++)
    {
    
    }
Client client = (Client) clientMap.get(clientNames[i]);
    try
    {
        client.stopClientOperation();
    }
    catch (RemoteException ex)
    {
        clientMap.remove(clientNames[i]);
    }
}

/**
 * Iterate over the list of clients and notify them to shut down.
 */
public void shutDownClients()
{
    String[] clientNames = new String[clientMap.size()];
    clientMap.keySet().toArray(clientNames);
    for (int i = 0; i < clientNames.length; i++)
    {
        Client client = (Client) clientMap.get(clientNames[i]);
        try
        {
            client.shutDownClient();
        }
        catch (RemoteException ex)
        {
            clientMap.remove(clientNames[i]);
        }
    }
}

/**
 * Select chromosomes for the next generation
 * @param probabilities the array of probabilities corresponding to
 * the chromosomes
 * @return the list of chromosomes selected for the next generation.
 */
private List selectChromosomes(double[] probabilities)
{
    List list = new LinkedList();
    double prob = 0;
    rawList.clear();
    for (int i = 0; i < 2 * populationSize; i++)
    {
        prob = Math.random();
        int j = 0;
        while (prob <= probabilities[j])
        {
            j++;
        }
        list.add( (Chromosome) processedList.get(j));
    }
    return list;
}

/**
 * Calculate the probabilities of each chromosome being carried over
 * to the next generation. This calculation is based on the roulette
* wheel selection method.
* @return the array of probabilities corresponding to the
* chromosomes.
*/
private double[] calculateProbabilities()
{
    double[] probs = new double[processedList.size()];
    double worst = ((Chromosome)processedList.getLast()).getFitness();
    double sum = 0;
    for (int i = 0; i < processedList.size(); i++)
    {
        sum += ( (Chromosome) processedList.get(i)).getFitness();
    }
    for (int i = 0; i < probs.length; i++)
    {
        probs[i] = (worst - ( (Chromosome)processedList.get(i)).getFitness())
            / sum;
    }
    return probs;
}

/**
 * Sort the given list using the chromosome comparator.
 * @param list the list to be sorted.
 * @return the sorted list.
 */
private List sortChromosomes(List list)
{
    synchronized (list)
    {
        Collections.sort(list, comparator);
    }
    return list;
}

}//EndOfClass ServerEngine
package edu.ohiou.ga;

/**
 * A class that extends the <code>java.lang.Exception</code> class to
 * process exceptions generated during the execution of the genetic
 * algorithm.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */

class GaException extends Exception
{
    /**
     * Default constructor that initializes the GaException.
     */
    public GaException()
    {
        super();
    }

    /**
     * Constructor that initializes the GaException with the given
     * message.
     * @param message the message to be displayed by the exception.
     */
    public GaException(String message)
    {
        super(message);
    }

    /**
     * Constructor that initializes the GaException with the given
     * message and cause.
     * @param message The message to be displayed by the exception.
     * @param cause The cause of the exception.
     */
    public GaException(String message, Throwable cause)
    {
        super(message, cause);
    }

    /**
     * Constructor that initializes the GaException with the given cause.
     * @param cause The cause of the exception.
     */
    public GaException(Throwable cause)
    {
        super(cause);
    }
}

//EndOfClass GaException
package edu.ohiou.ga;

/**
 * Class that holds various parameters of the genetic algorithm such as
 * the crossover and mutation probabilities, the number of generations
 * that the genetic algorithm has to run, and other such properties.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0 */

import java.io.Serializable;

public class GaParameters implements Serializable
{
    private double crossoverRate;
    private double mutationRate;
    private int populationSize;
    private int numberOfGenerations;

    /**
     * Constructor that creates the parameters
     * @param crossoverProb the crossover probability
     * @param mutationProb the mutation probability
     * @param populationSize the population size
     * @param numOfGens the number of generations
     */
    public GaParameters(double crossoverProb, double mutationProb,
                         int populationSize, int numOfGens)
    {
        crossoverRate = crossoverProb;
        mutationRate = mutationProb;
        this.populationSize = populationSize;
        numberOfGenerations = numOfGens;
    }

    /**
     * Returns the crossover probability of the GA
     * @return the crossover probability
     */
    public double crossoverProbability()
    {
        return crossoverRate;
    }

    /**
     * Returns the mutation probability of the GA
     * @return the mutation probability
     */
    public double mutationProbability()
{
    return mutationRate;
}

/**
 * Returns the population size of the GA
 * @return the population size
 */
public int populationSize()
{
    return populationSize;
}

/**
 * Returns the number of generations run for the GA
 * @return the number of generations
 */
public int numberOfGenerations()
{
    return numberOfGenerations;
}

}//EndOfClass GaParameters
package edu.ohiou.ga;

/**
* Data structure that stores the results of the genetic algorithm
* execution.
* @copyright Copyright (c) 2004
* @company Ohio University
* @author Nihar Shah
* @version 1.0
*/
import java.util.LinkedList;
import edu.ohiou.ga.interfaces.Chromosome;
import edu.ohiou.ga.interfaces.Generation;

public class GaResults
{
    /**
     * List that stores the results of each generation of the genetic
     * algorithm
     */
    private LinkedList results = new LinkedList();

    /**
     * Constructor that creates the result data structure
     */
    public GaResults()
    {
    }

    /**
     * Stores the results of the given generation in the data structure
     * @param gen the generation whose results are to be stored
     * @param genNo the generation number of the generation being stored
     */
    public void storeResults(Generation gen, int genNo)
    {
        int low = 0;
        int high = 0;
        double sum = 0.0;
        int size = gen.getPopulationSize();
        for (int i = 0; i < size; i++)
        {
            Chromosome ch = gen.getChromosome(i);
            if (ch.getFitness() < gen.getChromosome(low).getFitness())
            {
                low = i;
            }
            if (ch.getFitness() > gen.getChromosome(high).getFitness())
            {
                high = i;
            }
        }
    }
}
    sum += ch.getFitness();
}
ResultData rsd = new ResultData(gen.getChromosome(low),
    gen.getChromosome(high), genNo, sum / size);
results.addLast(rsd);

/**
 * Return the results of the generation specified by the generation number
 * @param genNo the generation number
 * @return the results associated with the generation
 */
public ResultData getResults(int genNo)
{
    return (ResultData) results.get(genNo);
}

} //EndOfClass GaResults
package edu.ohiou.ga;
/**
 * Data structure that stores the best and worst chromosomes, generation to
 * which they belong and the average fitness value of that generation.
 * @copyright Copyright (c) 2004
 * @company Ohio University
 * @author Nihar Shah
 * @version 1.0
 */
import edu.ohiou.ga.interfaces.Chromosome;

public class ResultData
{
    private Chromosome best;
    private Chromosome worst;
    private int generation;
    private double average;

    /**
     * Constructor that creates the result data for a given generation
     * @param best the best chromosome in the generation
     * @param worst the worst chromosome in the generation
     * @param generation the generation number of the generation
     * @param average the average value of chromosomes in the generation
     */
    public ResultData(Chromosome best, Chromosome worst, int generation,
            double average)
    {
        this.best = best;
        this.worst = worst;
        this.generation = generation;
        this.average = average;
    }

    /**
     * Return the best chromosome
     * @return the best chromosome
     */
    public Chromosome getBestChromosome()
    {
        return best;
    }

    /**
     * Return the worst chromosome
     * @return the worst chromosome
     */
    public Chromosome getWorstChromosome()
    {
        return worst;
    }
/**
 * Return the generation number
 * @return the generation number
 */
public int getGeneration()
{
    return generation;
}

/**
 * Return the average value of chromosomes
 * @return the average value of chromosomes
 */
public double getAverage()
{
    return average;
}

}//EndOfClass ResultData