A Hybrid Biologically Inspired Approach to Solving Job Shop Scheduling Problems

Jordan Frank

200020076

Table Of Contents

Outline

This project will consist of an implementation of two different biologically inspired methods which will be used together in an attempt to solve instances of a class of problems known as job shop scheduling problems. A job shop scheduling problem is concerned with assigning a set of discrete operations to a set of processors. The operations are grouped into subsets called jobs. For the purpose of this project, we will always make the number of operations per job the same for each job. The operations that make up a job must proceed in a specific order, and an operation can only proceed when its preceding operation has completed. Each operation requires exclusive access to a specific processor for a specific amount of time. The problem, then, is to assign each operation to a processor in a way that abides by these two constraints and minimizes the total time required to complete all of the jobs. Clearly the job shop scheduling problem is an example of a constraint-based problem.

The two biologically inspired methods that we will use to solve job shop scheduling problems are genetic algorithms and neural networks. A genetic algorithm is an algorithm based on a very rough approximation of how the evolutionary process has occurred in the animal kingdom. A solution to a problem is represented as a sequence of “genes” which form a “chromosome”. A large number of random chromosomes are generated, and then the quality of the solutions that they represent is evaluated. The chromosomes which represent the solutions that are of the highest quality then go on to “mate” to produce a new generation of chromosomes. The process repeats, and it is assumed that eventually, by only mating the high quality solutions, an optimal solution will be reached. This is the concept of “survival of the fittest”. A neural network is a processing structure that is made up of many small units which we refer to as nodes. Nodes are extremely simple processing units that simply take a number of inputs, sum them up, and then generate a single output based on this sum. The nodes are grouped into layers, and the layers are connected to each other by weighted connections. An input is provided to the input layer, and then the output signals propagate along the weighted connections to the next layer and so on until they reach the output layer. The output signal of the output layer represents the solution generated by the neural network. Based on this output, the neural network can be made to produce a different output by changing the weighting of the connections. Neural networks take advantage of what is called massive parallelism, in which a huge number of very simple processes operate in parallel and coordinate and cooperate in such a way that an extremely complex behaviour can emerge. Neural network have been shown to be extremely good at finding patterns in large sets of data, and it is this property that we will take advantage of for solving our job shop scheduling problems.

We will create a system that is a hybrid of these two biologically inspired systems, and we will apply our system to a set of job shop scheduling problems in order to evaluate it's performance. To measure the performance of the algorithm, we will compare the quality of solutions that it generates for a number of standard previously studied problems to the results generated by a number of other methods as described in the paper “Job Shop Scheduling by Local Search” by Vaessens, Aarts, and Lenstra[1]. We will also include measurements of the running time required by the system to generate high quality solutions, as well as the rate at which the solutions improve as the algorithm proceeds. We will also experiment with various parameters for both the genetic algorithm and the neural network, and discuss the effects.

1 R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. “Job Shop Scheduling by Local Search.” *COSOR Memorandum 94-055*. Eidenhoven University of Technology, Eidenhoven, The Netherlands.

This approach to solving job shop scheduling problems is very interesting because of it's basis on biologically inspired methods. It is assumed that the evolutionary process has produced highly intelligent organisms by performing very simple operations, the combining of chromosomes during sexual reproduction, over a huge number of generations, and by providing an environment in which only the most fit organisms survive. By mimicking this process in software, with genetic algorithms, scientists have introduced an entirely new paradigm to the realm of software development. Instead of analyzing a problem in an iterative, straightforward manner, and then creating a specific solution based on the analysis, a generic evolutionary algorithm is applied to the problem, and the computer itself generates a solution. Often it has been the case that the solution that the computer comes up with is better than that created by a human. It has also been the case from time to time, that solution generated by the computer will work, but no one is able to figure out why it does. This is an exciting field, and it seems very likely that extremely interesting results will come of it. Neural networks, on the other hand, are based on our current understanding of how the brain processes information. It is believed that the brain consists of billions of neurons which are very simple processing devices. By wiring these neurons together with weighted synaptic connectors, and by propagating electrical impulses through the neurons, we get an incredibly advanced device, namely the human brain. That such tremendous processing power can arise from such simple components is very cool.

System Design

The system will be implemented using a combination of Python 2.3[2] and C++. Python will be used to quickly implement the framework for reading in problem sets and manipulating the data in such a way that it can be processed by the algorithm. Python will also be used to create a very basic GUI, using the wxPython[3] module. The GUI will allow the user to load a data set, manipulate the parameters for the algorithms, and view the output of the algorithms in a intuitive format both during the running of the system, and once the system has found it's “best” solution. Python has been chosen for these parts since they are not computationally intensive, and so an interpreted language allows for rapid implementation of the framework and GUI. For the algorithms, C++ will be used. Python allows for integration of modules that are implemented using compiled modules, so the framework can make use of the high performance compiled C++ modules. The system will be implemented on a Macintosh system running OS 10.3 using GNU emacs as an editor, GCC 3.3 as the compiler for the C++ code, and the Python 2.3 interpreter as the runtime environment. By using Python and wxPython for the GUI, the framework will be platform independent, though the C++ modules will have to be compiled under different operating systems, In theory the system will run on Macintosh, Windows, or Linux without modification by simply compiling the modules on the target system.

The framework will be highly modular so that in the future, different algorithms can be “plugged in”. The structure of the framework is shown in Figure 1. We will look at each component in detail.

Problem Set

The problem set is simply a flat text file that specifies the job shop problem. The

2 http://www.python.org/
3 http://www.wxpython.org/

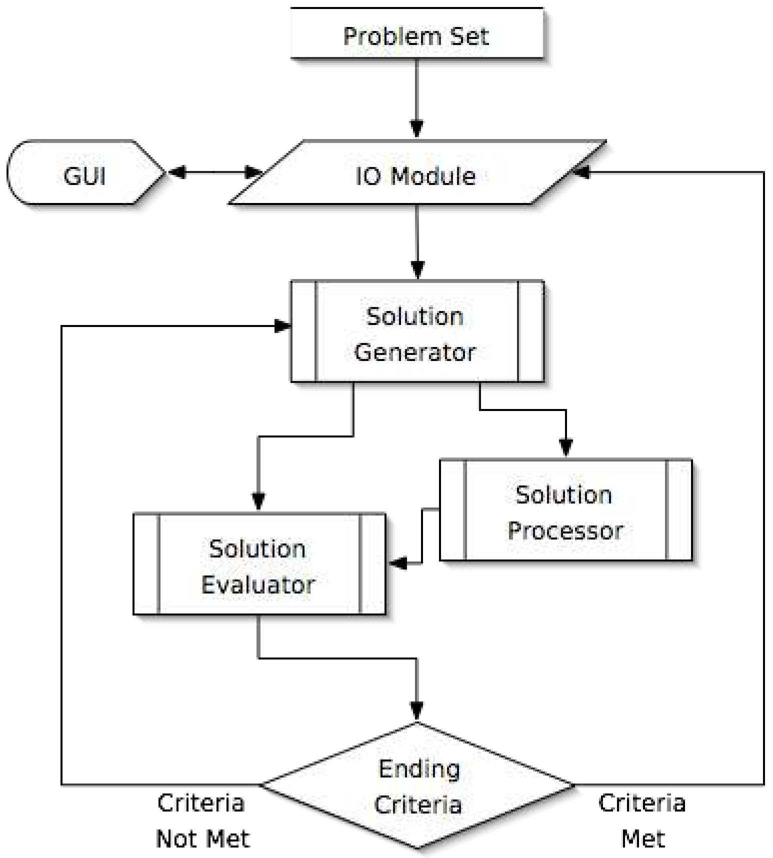


Figure 1: System Diagram

first line of the file states the number of jobs and the number of machines, and then it is followed by one line for each job, listing the machine number and processing time for each step of the job. This data is read from the file system and processed by the IO Module.

IO Module

The IO module is responsible for reading in the data from the problem set and communicating with the GUI. The problem set is read in and converted into a data structure representing the problem, which is then passed on to the solution generator. Input from the GUI representing the parameters for the algorithm are also passed on to the solution generator. When the system has decided on a solution, that solution is fed back into the IO module where it is outputted to the GUI.

GUI

The GUI is the interface through which the user interacts with the system. It consists of a set of input boxes for setting the various parameters such as the population rate, the mutation rate, the survival rate, and the average offspring per parent ratio for the genetic algorithm and the learning rate for the neural network. The solution that the system generates will be displayed as a chart displaying the time line and the various machines with boxes representing the job that is active on the specific machine at the specific time.

Solution Generator

The solution generator consists of the genetic algorithm. The genetic algorithm initially takes the problem set from the IO module and generates a number of random solutions. The solutions are then fed to the solution evaluator. After the solutions get evaluated, if the ending criteria is not met, the solutions are fed back into the solution generator along with their quality rating, and a new set of solutions are generated based on this information. After a certain number of iterations of this type, the solutions are then fed into the solution processor, as well as the solution evaluator.

Solution Processor

The solution processor consists of the neural network. After a certain iterations of the genetic algorithm, the solutions begin to flow into the solution processor. These solutions are then used to train the neural network. After each training iteration, the neural network is used to generate a new solution, which is then fed into the solution evaluator. Therefore, after the neural network begins processing, it contributes a new solution to the set of solutions, and this new solution then competes with the other solutions generated by the genetic algorithm.

Solution Evaluator

The solution evaluator takes a set of solutions and assigns a value to each solution that represents the makespan of that solution. The makespan is equivalent to the total time that the schedule represented by the solution would take to run to completion. The higher the makespan, the lower the quality of the solution, and so the system will be designed with the goal of minimizing the makespan.

Ending Criteria

The ending criteria specifies when the system will stop running and decide on a solution. The criteria may be based on the quality of the solution, the number of iterations that the system has gone through, or both. If the criteria is not met, the solutions and their corresponding makespans are fed back into the solution generator, and if the criteria is met then the solution with the smallest makespan will be fed to the IO module so that it can be displayed in the GUI.

The algorithm will make use of a number of data structures. The number of The problem set will be internally represented by a set of two matrices, the width of which is the maximum number of operations per job, the height of which is the total number of jobs. The elements of the first matrix represent the resource required for the specific operation, and so the element at position (*i*, *j*) represents the resource required by the *i*-th operation of job *j*. The elements of the second matrix represent the processing time required by each specific operation and ,similar to the first matrix, the element at position (*i*, *j*) represents the time required by the *i*-th operation of job *j*. Because zeros play a crucial role in the data structures and the algorithm, as we will see later, the machines, jobs, and operations are all indexed starting with one, so for example, the first job is Job1, not Job0. The solutions are represented as a vector of integer values. The length of the vector is equal to the number of operations per job multiplied by the number of jobs. The elements of the vector represent the job that is to be scheduled, and the order of the elements specifies the order in which the jobs are scheduled. So for example if we have the vector [2, 4, 5, 3, 5, 1], this specifies that we will first assign the first unassigned operation of job 2 to the resource that it requires, then we will assign the first unassigned operation of job 4, then job 5, then job 3, then job 5, and finally job 1. The order is important, because it may be the case that both jobs 2 and 4 require the same resource for their first operation. In this example, job 2 would be assigned the resource, and job 4 would have to wait until job 2 is done with that resource before it can begin it's operation. A vector data structure was chosen for two reasons. First of all, both genetic algorithms and neural networks are designed to operate on a sequence of elements. For genetic algorithms, this sequence represents the chromosome, and its elements represent the individual genes. In keeping with our biological analogue, the numeric values representing the resources comprise the set of alleles, which are the set of possible values for the genes. With neural networks, the input is usually given as a bit vector, and so with a little coaxing, our vector can be used. The second reason for using a vector for our solution is that it is essentially a recipe for building a schedule, and as such it will always represent a feasible, or valid, solution. We never have to worry about ensuring that our constraints are satisfied as long as the method for generating the schedule from the solution vector is implemented properly.

The algorithm for generating a schedule from a solution consists of iterating over the elements of the solution vector and constructing the schedule from beginning to end. Due to the two constraints, that the operations must proceed in order, and that no two operations can make use of a single resource at the same time, we have not been able to come up with an algorithm that consists of only a series of matrix operations. We do suspect though, that such an algorithm, which would operate in constant time regardless of the size of the inputs, may exist. Our current algorithm is $O(n^2)$ since on each iteration, we must find the earliest time that the resource in question is free, as well as the earliest time that the preceding operation has completed, and then take the maximum of the two as the start time for the current operation. In order to experiment with various solution vectors, we have already implemented this algorithm as an OpenOffice[4] spreadsheet (which also happens to be compatible with Excel, but Microsoft is evil). The solution vector is entered into the spreadsheet as an array, and then the spreadsheet automatically calculates the start and end time of each operation, calculates the makespan for the schedule, and generates a visual representation of the schedule. This makes it easy to see how modifications to the solution vector affect the final schedule. A sample solution for a simple 6-job, 6-machine, 6-operations/machine problem is included in Appendix A.

Work Plan

This system must be completed by December 1, 2004. This project plan was completed on November 1, 2004, and so we have exactly one month to complete the system. As stated earlier, Python has been chosen as the language in which the framework will be implemented in order to speed up development, and leave adequate time for working on the real guts of the system, the algorithms. Here is the breakdown, in hours of the time required for each of the components described in the previous section, as well as the number of days over which this work will be spread.

- IO Module and GUI – 6 hours over 2 days.
- Solution Generator – 15 hours over 4 days.

4 http://www.openoffice.org/ - OpenOffice has evolved into an excellent alternative to Microsoft's Office suite, and is nearly 100% compatible. It also happens to be free.

- Solution Evaluator – 2 hours over 1 day.
- Solution Processor – 20 hours over 4 days.

We will also budget 2 hours over 1 day for wiring together the components, 4 hours over 1 day for testing, and 12 hours over two days for evaluation and reporting. 4 hours for testing would typically be inadequate, but due to the rapid development of this project, as well as the modularity, the project will be tested throughout the entire development process. Figure 2 shows how this time will be allocated throughout the next month. As this project is being done by a single student, project management will require very little, if any, time, and all tasks will be allocated to a single developer. The total amount of time required for this project will be approximately 61 hours, and just to bring the real world into this, a project such as this would cost around $15,000 were you to hire the company that I work for (and co-founded) to develop it.

Testing and Evaluation

In order to evaluate this project, we will be comparing the output to that of a number of other local search algorithms on a set of problems that are widely used for testing job shop scheduling algorithms. The system will be tested on the 82 job shop scheduling problems provided by the instructor[5]. For the problems that were also

November

	Sun.	Mon.	Tues.	Wed.	Thur.	Fri.	Sat.
		1	2	3	4	5	6
IO Module and GUI					▨	▨	
Solution Generator							▨
Solution Evaluator							
Solution Processor							
Component Integration							
Testing and Bugfixes							
Evaluation and Reporting							
	7	8	9	10	11	12	13
IO Module and GUI							
Solution Generator	▨	▨	▨				
Solution Evaluator					▨		
Solution Processor							▨
Component Integration							
Testing and Bugfixes							
Evaluation and Reporting							
IO Module and GUI	14	15	16	17	18	19	20
Solution Generator							
Solution Evaluator							
Solution Processor	▨	▨	▨				
Component Integration					▨		
Testing and Bugfixes							▨
Evaluation and Reporting							
IO Module and GUI	21	22	23	24	25	26	27
Solution Generator							
Solution Evaluator							
Solution Processor							
Component Integration							
Testing and Bugfixes	▨						
Evaluation and Reporting			▨		▨		

Figure 2: Timeline for Project

5 http://www.cs.sfu.ca/CC/417/havens/project/jobshop1.txt

tested in the Vaessens, Aarts, and Lenstra paper, we will provide a detailed comparison of our results and theirs. We will also provide graphs showing the rate at which the solutions to specific problems improve over the course of the operation of the system, as well as a discussion of how the various parameters for the algorithms affect the quality of the solution and the amount of time required to find the best solutions.

Appendix A: Sample Mapping from Solution Vector to Schedule

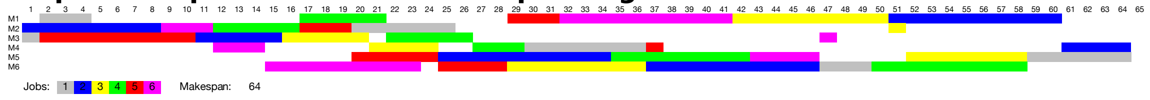
Fisher and Thompson 6x6 Instance (also known as mt06)

$$\underline{F}\begin{vmatrix} 3 & 1 & 2 & 4 & 6 & 5 \\ 2 & 3 & 5 & 6 & 1 & 4 \\ 3 & 4 & 6 & 1 & 2 & 5 \\ 2 & 1 & 3 & 4 & 5 & 6 \\ 3 & 2 & 5 & 6 & 1 & 4 \\ 2 & 4 & 6 & 1 & 5 & 3 \end{vmatrix} \qquad \underline{P}\begin{vmatrix} 1 & 3 & 6 & 7 & 3 & 6 \\ 8 & 5 & 10 & 10 & 10 & 4 \\ 5 & 4 & 8 & 9 & 1 & 7 \\ 5 & 5 & 5 & 3 & 8 & 9 \\ 9 & 3 & 5 & 4 & 3 & 1 \\ 3 & 3 & 9 & 10 & 4 & 1 \end{vmatrix}$$

Sample Solution Vector

[2,3,2,4,6,3,6,4,1,3,4,3,1,6,2,6,3,5,6,5,1,4,1,2,3,6,5,4,5,5,2,1,2,5,4,1]

Graphical Representation of corresponding Schedule


M1
M2
M3
M4
M5
M6
Jobs: 1 2 3 4 5 6
Makespan: 64