

# **Scheduling Algorithms for the Open-Shop Problem**

Matthew Norman

Computer Science

2002/2003

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) \_\_\_\_\_

## **Project Summary**

The aim of this project was to implement a software component to solve the open-shop scheduling problem and analyse its capabilities and efficiency. The software component was designed to use the modelling concepts of the library of scheduling algorithms LiSA. As a result of this design feature the possibility of incorporating the software component into the LiSA software itself was enabled.

## **Acknowledgements**

Thank you to my project supervisor Dr Natasha Shakhlevich for giving me help and direction on this project.

Also thanks to Graham Hardman for his help in the installation and compilation of LiSA.

# Contents

<b>Project Summary</b> .....	<b>ii</b>
<b>Acknowledgements</b> .....	<b>iii</b>
<b>Contents</b> .....	<b>iv</b>
<b>Chapter 1 – Project Overview</b> .....	<b>6</b>
1.1 Introduction .....	6
1.2 Objectives .....	6
1.3 Minimum Requirements .....	7
1.4 Deliverables .....	7
1.5 Project Schedule .....	8
1.6 Project Milestones .....	8
<b>Chapter 2 – Background Research</b> .....	<b>10</b>
2.1 The Open-Shop .....	10
2.2 Existing Open-Shop Algorithms .....	11
2.2.1 The LPT Rule .....	11
2.2.2 Gonzalez and Sahni’s Algorithm .....	12
2.3 Pinedo’s Rules .....	13
2.3.1 The LAPT Rule .....	13
2.3.2 The LTRPOM Rule .....	15
2.4 LiSA – Library of Scheduling Algorithms .....	15
2.5 LiSA Modelling Concepts .....	16
2.6 LiSA External Module Development .....	18
2.7 Possible Project Enhancements .....	19
<b>Chapter 3 – Algorithm Design</b> .....	<b>21</b>
3.1 Choice of Platform and Language .....	21
3.2 Algorithm Breakdown .....	22
3.2.1 LiSA Objects .....	22
3.2.2 Set of Data .....	22
3.2.3 Main Algorithm .....	23
3.2.3.1 Free .....	24
3.2.3.2 Calculate Schedule .....	24
3.2.3.3 Machine Available .....	24
3.2.3.4 Calculate Machine .....	24
3.2.3.5 Machine Sum .....	25
3.2.3.6 Calculate Job .....	25
3.2.3.7 Job Sum .....	25
3.3 Evaluation Criteria .....	26
<b>Chapter 4 – Algorithm Implementation</b> .....	<b>27</b>
4.1 Input and Output .....	27
4.2 LiSA Objects .....	27
4.3 Data Structures .....	28
4.4 Main Algorithm .....	29
4.4.1 free() .....	29

4.4.2 calculateSchedule()	29
4.4.3 machineAvailable()	30
4.4.4 calculateMachine()	30
4.4.5 machineSum()	30
4.4.6 calculateJob()	30
4.4.7 jobSum()	31
4.5 Compilation and Installation	31
4.6 Testing	32
<b>Chapter 5 – Implementation Issues</b>	<b>33</b>
5.1 Introduction	33
5.2 LiSA Compilation	33
5.3 German	34
5.4 Select First Current Machine	34
5.5 Select First Current Job	35
<b>Chapter 6 – User Guide</b>	<b>36</b>
6.1 Introduction	36
6.2 Compilation and Installation	36
6.3 Executing from within LiSA	37
6.4 Executing from the Command Line	38
<b>Chapter 7 – Computational Experiments</b>	<b>39</b>
7.1 Introduction	39
7.2 Open-Shop Instances	39
7.3 Algorithms Used	39
7.4 Results	40
<b>Chapter 8 – Evaluation</b>	<b>41</b>
8.1 Minimum Requirements	41
8.2 Software Component	41
8.3 Project Schedule	43
<b>Chapter 9 – Conclusions and Future Work</b>	<b>44</b>
9.1 As It Is	44
9.2 As It Could Be	44
<b>References</b>	<b>45</b>
<b>Appendix A – Project Reflection</b>	<b>47</b>
<b>Appendix B – Example ALG Files</b>	<b>48</b>
<b>Appendix C – Example LSA Files</b>	<b>50</b>

# **Chapter 1**

## **Project Overview**

### **1.1 Introduction**

Scheduling is present all around us in everyday life, whether it be your local bus timetable, programmes executing on a computer or the phases in a construction project. In fact, scheduling can be defined in a much more general manner to encompass all of these forms. According to Pinedo (2002:1), scheduling is the assignment of resources to tasks with the target of optimising certain objectives.

As scheduling is so widespread it can be important to make use of software packages to create, solve and explore certain scheduling problems. One such package is LiSA (the Library of Scheduling Algorithms), which provides, among other things, some pre-compiled algorithms for solving certain scheduling problems.

This project aimed to create a software component to solve the open-shop problem based on LiSA's modelling concepts and Pinedo's LAPT and LTRPOM rules.

### **1.2 Objectives**

There are a number of project objectives that must be completed to ensure that the project aim is successfully completed:

1. Study the open-shop problem.
2. Study Pinedo's LAPT and LTRPOM rules.
3. Study the modelling concepts of LiSA.
4. Combine steps 1, 2 and 3 to design a software component that solves the open-shop problem using Pinedo's rules and the modelling concepts of LiSA.
5. Implement the design from step 4 to make an executable software component and produce a small user-guide.
6. Test the software component made in step 5 to ensure it operates correctly. If successful, proceed to step 7. If not, identify and fix the problem and re-test.

7. Devise a set of instances of open-shop problems using random number generation to be used in the computational experiments.
8. Execute the software component and a selection of the known algorithms from LiSA on the instances devised in step 7. Note the results.
9. Analyse the performance of the software component against the known algorithms by comparing the results obtained from step 8.
10. Draw conclusions on the capabilities and efficiency of the software component compared to the known algorithms from LiSA.

### **1.3 Minimum Requirements**

From these objectives it is possible to devise a set of minimum requirements that must be reached in order to deliver a solution to the problem:

1. To implement Pinedo's rules as a software component using the modelling concepts of LiSA.
2. To learn and analyse the known algorithms for the open-shop problem.
3. To perform computational experiments to analyse the performance of the software component against the known algorithms from the library of scheduling algorithms LiSA.

### **1.4 Deliverables**

When completed, the project will have a number of deliverables:

1. The software component.
2. A small user-guide for the software component.
3. The project report.

## **1.5 Project Schedule**

Each objective has a time period for completion associated with it. The periods of Christmas, New Year and the end of first semester exams and their revision time must also be taken into account. The project schedule can be represented in the form of a Gantt chart (see Figure 1.1).

## **1.6 Project Milestones**

From the project objectives it is possible to identify three milestones for progress:

1. The completion of the study of the open-shop, the LAPT and LTRPOM rules and the modelling concepts of LiSA. All of the necessary theory has been gathered and the software component can now be considered.
2. The completion of the design, creation and testing of the software component. The key ingredient of the project has been finished and the computational experiments can begin.
3. The completion of the computational experiments, analysis of the performance and conclusion on the capabilities and efficiency. The aim of the project has been attained.



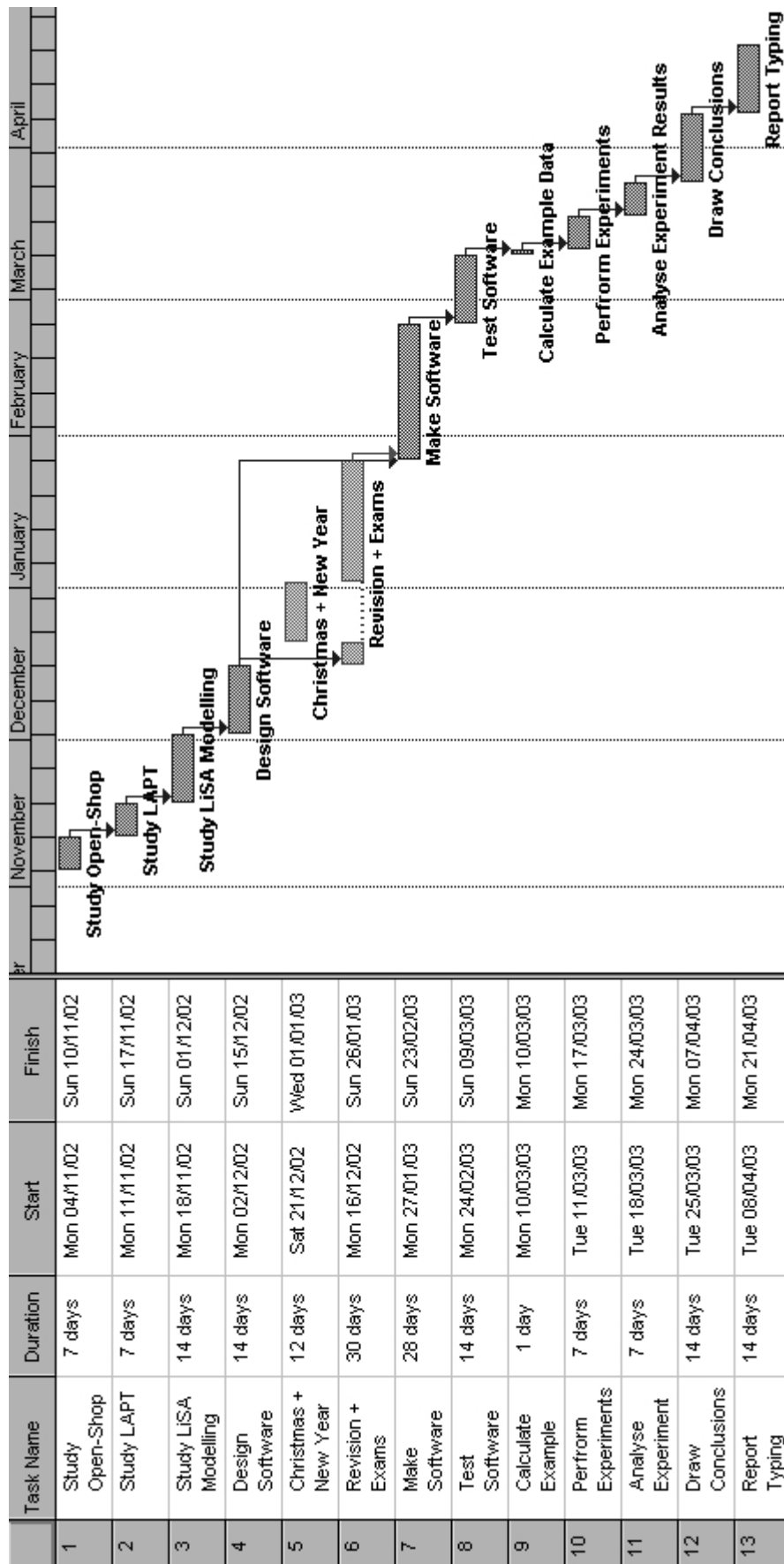


Figure 1.1 – Gantt Chart of Project Schedule

## **Chapter 2**

### **Background Research**

#### **2.1 The Open-Shop**

An open-shop is a multi-operation model where the routes of the jobs are open, i.e. the routes are not predetermined and the jobs can be sequenced in any order on each machine.

An open-shop problem can be described in a special notation that is used for all scheduling problems. This notation uses a triplet  $\alpha \mid \beta \mid \gamma$ , where  $\alpha$  holds the machine environment,  $\beta$  holds processing characteristics and constraints and  $\gamma$  holds the objective function. In the case of this project, the machine environment will be an open-shop, which is denoted by a single O followed by the number of machines in the model (or none at all for an arbitrary number of machines). The processing characteristics and constraints will not apply to this project but could specify that release dates for jobs are used or preemptions are allowed. The objective function of an open-shop model can be one of many. An example is the total completion time ( $\sum C_j$ ), which is the sum of the completion times of all jobs, but this project will be considering the makespan ( $C_{\max}$ ), which is the completion time of the last job to leave the system.

There are 3 main types of algorithms that can be used to solve open-shop problems. These are exact algorithms, approximation algorithms and heuristic algorithms.

Exact algorithms can find optimal solutions for particular problems, for example, branch and bound algorithms. Approximation algorithms produce solutions in polynomial time that are guaranteed to be within a fixed percentage of the actual optimum. Heuristic algorithms produce feasible solutions that are not guaranteed to be close to the optimum. Heuristic algorithms can be split into two classes: construction heuristics and improvement heuristics. Construction heuristics start without a schedule and add one job at a time whereas improvement heuristics start with a schedule and try to find a better 'similar' schedule.

## **2.2 Existing Open-Shop Algorithms**

### **2.2.1 The LPT Rule**

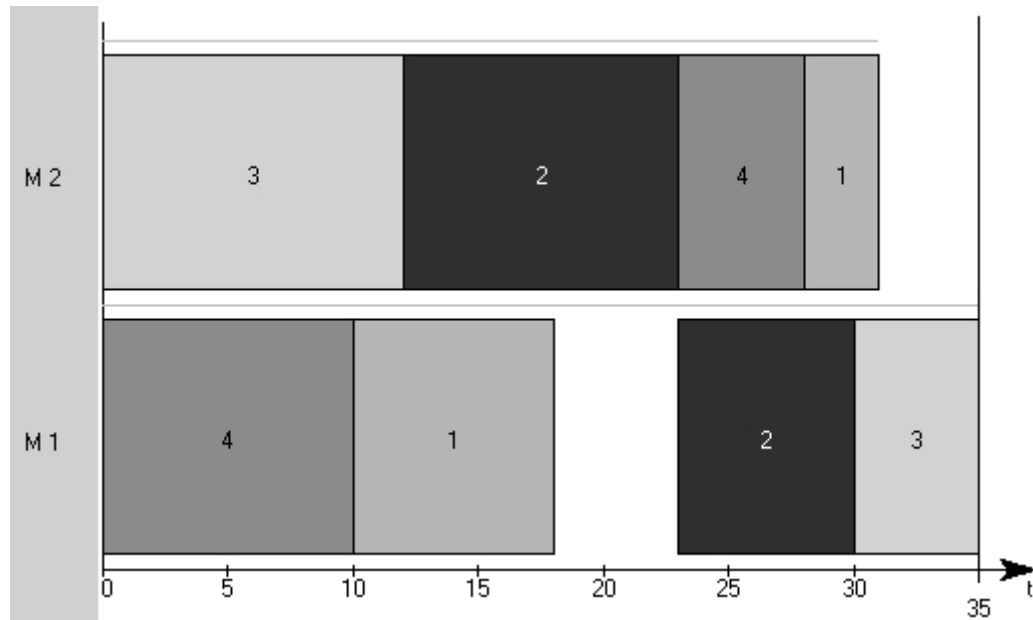
An existing construction heuristic algorithm to use on an open-shop problem uses the LPT (Longest Processing Time first) rule. This rule states that whenever a machine is freed, start processing the job with the largest remaining processing time on that machine.

Here is an example of the LPT rule used on an open-shop problem with 2 machines and the makespan as the objective function,  $O2 \parallel C_{\max}$  :

jobs	1	2	3	4
$p_{1j}$	8	7	5	10
$p_{2j}$	3	11	12	5

- ◆ First we consider machine 1
  - The job with the largest processing time on this machine is job 4, with 10.
  - Job 4 is scheduled first on machine 1.
- ◆ Now we consider machine 2.
  - The job with the largest processing time on this machine is job 3, with 12.
  - Job 3 is scheduled first on machine 2.
- ◆ Now we consider machine 1 again as it becomes free before machine 2.
  - The job with the largest processing time on this machine is job 1, with 8.
  - Job 1 is scheduled second on machine 1.
- ◆ Now we consider machine 2 again as it next to be free.
  - The job with the largest processing time on this machine is job 2, with 11.
  - Job 2 is scheduled second on machine 2.
- ◆ Now we consider machine 1 again as it next to be free.
  - The job with the largest processing time on this machine is job 2, with 7.
  - Job 2 is scheduled third on machine 2, but it cannot start processing on machine 1 until it has finished processing on machine 2.
  - This results in machine 1 being left idle for a period of 5 units.
- ◆ Now we consider machine 2 again as it next to be free.
  - The job with the largest processing time on this machine is job 4, with 5.
  - Job 4 is scheduled third on machine 2.

This process continues until there are no more jobs left to schedule on either machine and it results in the following schedule represented as a Gantt chart:



**Figure 2.1 – Gantt Chart of Schedule Constructed with LPT Rule**

This schedule gives a non-optimal makespan,  $C_{\max} = 35$ . It is described as non-optimal because the makespan is being affected by the idle time on machine 1, causing the completion time of the last job to leave the system to be later than necessary. If, for example, we moved job 3 on machine 1 in front of job 2 we would dispose of the idle time and be presented with an optimal makespan.

### **2.2.2 Gonzalez and Sahni's Algorithm**

Gonzalez and Sahni (1976) developed an exact algorithm for the open-shop problem with two machines and the makespan as the objective function,  $O2 \parallel C_{\max}$ . Not only did it guarantee an optimal schedule, but it was a linear time algorithm as well.

As well as developing this algorithm, Gonzalez and Sahni (1976) went on to prove that an open-shop problem with more than two machines and the makespan as the objective function is NP-complete,  $Om \parallel C_{\max}$  with  $m \geq 3$ .

## **2.3 Pinedo's Rules**

Pinedo (2002) has developed two rules for solving the open-shop problem: the Longest Alternate Processing Time first (LAPT) rule and the Longest Total Remaining Processing on Other Machines first (LTRPOM) rule. The LAPT rule is an exact algorithm for the open-shop problem with two machines and the makespan as the objective function,  $O2 \parallel C_{\max}$ . The LTRPOM rule is a construction heuristic for the open-shop problem with more than two machines and the makespan as the objective function,  $Om \parallel C_{\max}$  with  $m \geq 3$ . Pinedo (2002) comments that the LAPT rule can actually be considered as a unique case of the more general LTRPOM rule.

### **2.3.1 The LAPT Rule**

Pinedo (2002) describes his LAPT rule as follows:

Whenever a machine is freed, start processing among the jobs that have not yet received processing on either machine the one with the longest processing time on the other machine.

(Pinedo 2002:188)

Here is an example of the LAPT rule used on the same problem in the LPT rule example above,  $O2 \parallel C_{\max}$  :

- ◆ First we consider machine 1.
  - The longest processing time on the alternate machine, machine 2, is job 3 with 12.
  - Job 3 is scheduled first on machine 1.
- ◆ Now we consider machine 2.
  - The longest processing time on the alternate machine, machine 1, is job 4 with 10.
  - Job 4 is scheduled first on machine 2.
- ◆ Both machines now become free at the same time, so a machine is selected arbitrarily.
- ◆ In this case we consider machine 1 again.

- The longest processing time on the alternate machine, machine 2, is job 2 with 11.
- Job 2 is scheduled second on machine 1.
- ◆ Now we consider machine 2 again as it next to be free.
  - The longest processing time on the alternate machine, machine 1, is job 1 with 8.
  - Job 1 is scheduled second on machine 2.
- ◆ Now we consider machine 2 again as it next to be free.
  - The longest processing time on the alternate machine, machine 1, is job 2 with 7, but we cannot schedule job 2 next on machine 2 because it was the last operation to be scheduled on machine 1 and is still being processed.
  - Scheduling job 2 next on machine 2 would result in *unforced* idleness on machine 2 as we have another job available to us.
  - As a result, job 3 is scheduled third on machine 2.
- ◆ Now we consider machine 1 again as it next to be free.
  - The longest processing time on the alternate machine, machine 2, is job 4 with 5.
  - Job 4 is scheduled third on machine 1.

This process continues until there are no more jobs left to schedule on either machine and it results in the following schedule:

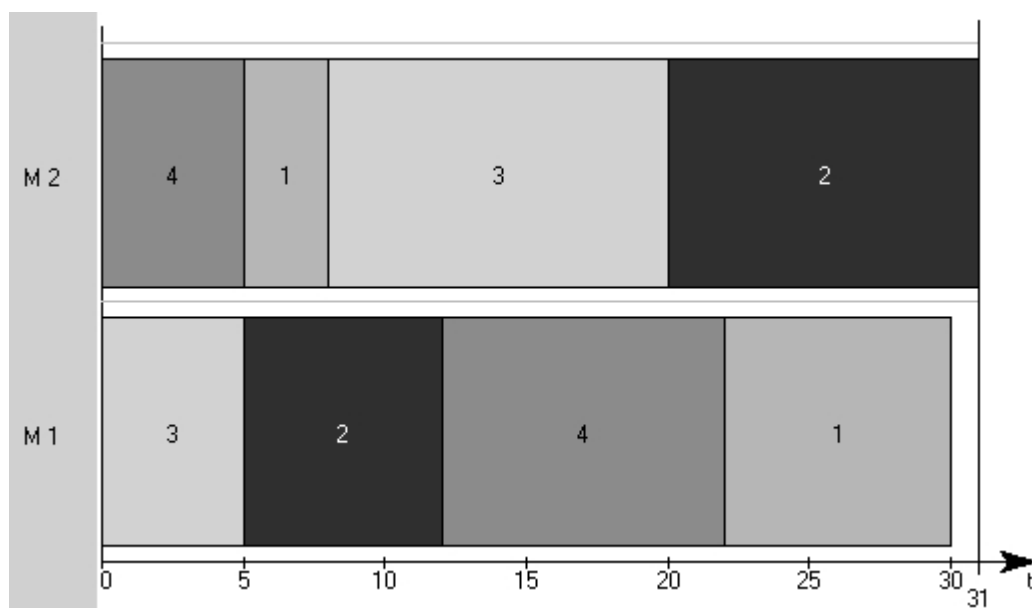


Figure 2.2 – Gantt Chart of Schedule Constructed with LAPT Rule

This schedule gives an optimal makespan,  $C_{\max} = 31$ . It is described as optimal because the makespan cannot be minimised any further as it is not being affected by any idle times. This can be seen on machine 2, where there are no idle periods and the makespan is equal to the sum of processing times of the jobs on machine 2.

It is not a surprise that the LAPT rule gives us an optimal makespan for this problem because it is an exact algorithm for the open-shop problem with two machines and the makespan as the objective function. Pinedo (2002) also proves that the LAPT rule produces an optimal schedule for  $O2 \parallel C_{\max}$ .

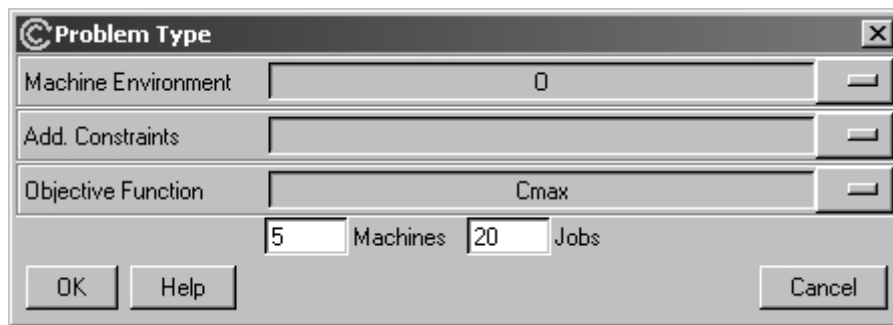
### **2.3.2 The LTRPOM Rule**

As mentioned earlier, the LAPT rule can actually be considered as a unique case of the more general LTRPOM rule, which states that whenever a machine is freed, start processing the job with the largest total remaining processing time on all other machines.

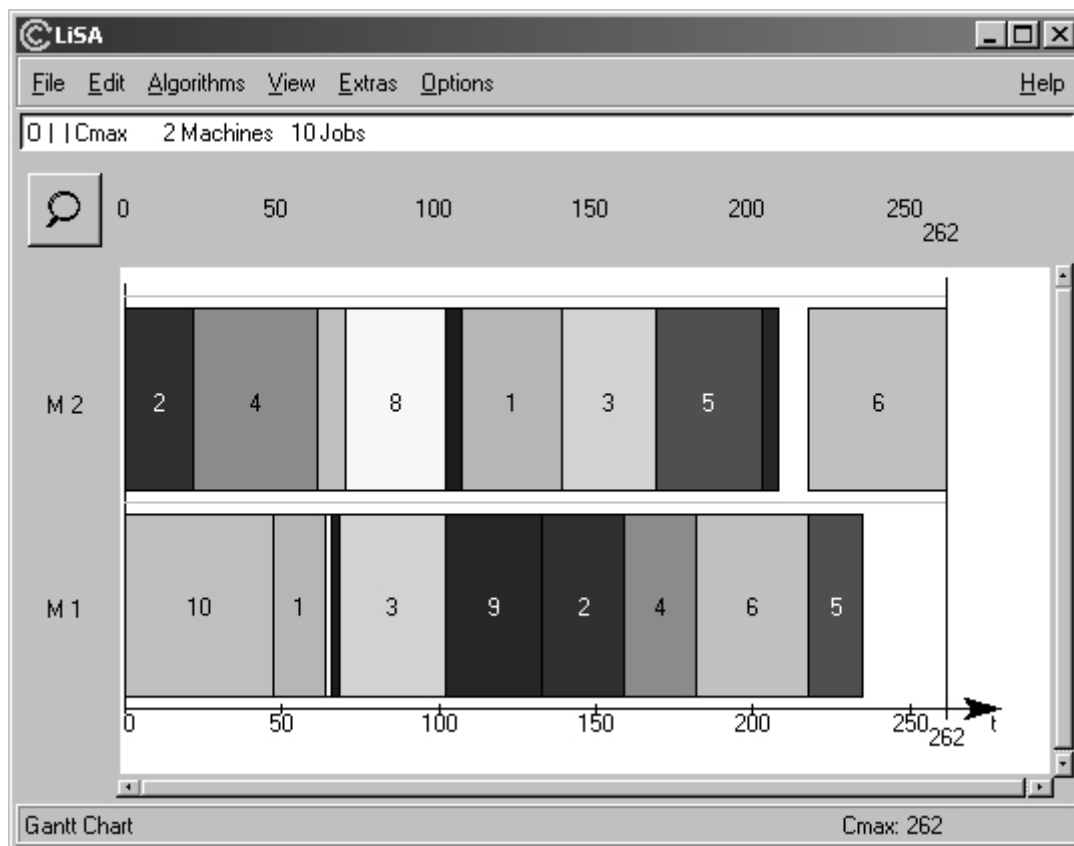
If the LTRPOM rule is used on the problem  $O2 \parallel C_{\max}$ , it behaves as the LAPT rule and produces an optimal schedule. However, the LTRPOM rule cannot guarantee an optimal schedule for the problem  $Om \parallel C_{\max}$  with  $m \geq 3$  as this problem is NP-complete, as proved by Gonzalez and Sahni (1976) and Pinedo (2002).

## **2.4 LiSA - Library of Scheduling Algorithms**

The LiSA-Homepage [Online] describes LiSA as software for solving and examining scheduling problems. It uses a graphical environment incorporating algorithms and tools for displaying and manipulating schedules. LiSA also supports a wide range of machine environments, processing characteristics and constraints and objective functions (figure 2.3) giving it a thorough coverage of scheduling problems.



**Figure 2.3 – Definition of a Scheduling Problem in LiSA**



**Figure 2.4 – The Main LiSA Interface**

LiSA is intended for use in research for fast implementation and testing of algorithms, in teaching for examining properties of existing algorithms and in development for the design and implementation of scheduling algorithms for practical use (LiSA-Homepage [Online]). LiSA is available for Microsoft Windows and Linux platforms.



## **2.5 LiSA Modelling Concepts**

Bräsel (1996) notes that the fundamental modelling concept used in LiSA for the modelling of shop problems and their solutions is that of the Latin rectangle. Bräsel goes on to describe a Latin Rectangle as an (n x m) matrix with each element coming from the set of all job numbers where each job number only appears at most once in each row and each column.

Note: n denotes the number of jobs and m denotes the number of machines.

A shop problem and its solution modelled in LiSA would use a Latin rectangle to hold a variety of data such as processing times of each job on each machine; completion times of each job on each machine; and the position of the execution of each job on each machine. For example, a Latin rectangle holding the schedule in the above LAPT rule example would be:

$$LR = \begin{bmatrix} 4 & 2 \\ 2 & 4 \\ 1 & 3 \\ 3 & 1 \end{bmatrix}$$

It has 4 rows for the 4 jobs and 2 columns for the 2 machines. The first row shows that job 1 is executed 4<sup>th</sup> on machine 1 and 2<sup>nd</sup> on machine 2. The second row shows that job 2 is executed 2<sup>nd</sup> on machine 1 and 4<sup>th</sup> on machine 2. The third row shows the positions of job 3 on both the machines and so on for each row.

Another example of a Latin rectangle holding the completion times of each job in the above LAPT rule example would be:

$$CIJ = \begin{bmatrix} 30 & 8 \\ 12 & 31 \\ 5 & 20 \\ 22 & 5 \end{bmatrix}$$

It again has 4 rows for the 4 jobs and 2 columns for the 2 machines. The first row shows that job 1 completes at time 30 on machine 1 and at time 8 on machine 2.

The second row shows that job 2 completes at time 12 on machine 1 and at time 31 on machine 2. The third row shows the completion times of job 3 on both the machines and so on for each row.

It is also possible to calculate the value of the makespan objective function from a Latin rectangle holding the completion times of each job; it is simply the largest value present.

## **2.6 LiSA External Module Development**

To create an external module and incorporate it into the LiSA software itself there are a number of rules that must be obeyed and conventions that must be adopted.

Firstly, you must specify which problem(s) your algorithm can be applied to either as an exact or heuristic technique and then define whether it is a constructive or an iterative algorithm and which parameters it takes (Dornheim 2003:5). This classification is held in an ALG file, which is the way LiSA addresses algorithm descriptions. A number of examples of ALG files can be found in Appendix B.

Once the specification is complete the actual algorithm can be implemented. If written in C++, the external module can make use of LiSA's data structures, file access techniques and compilation and installation methods.

LiSA provides a number of previously implemented classes written in C++ header files that can be included in external module development so as to make use of LiSA's data structures and file access techniques. A brief summary of some of the main files is shown below:

<b>C++ File</b>	<b>Use</b>
global.hpp	Holds data on the problem type including machine environment, constraints and the objective function.
matrix.hpp	Contains vectors and matrices with dynamic size and fast access.

ctrlpara.hpp	Necessary for parsing the control parameters in the input file (see below).
schedule.hpp	Necessary for writing the resulting schedule into an output file.
ptype.hpp	Necessary for parsing the problem type in the input file.
lvalues.hpp	Necessary for parsing the Latin rectangle values in the input file.
except.hpp	Holds a class for storing error messages and error codes.

Note: Control parameters will not be needed in the implementation of the LAPT and LTRPOM rules. An example of when control parameters are needed is in the execution of the branch and bound algorithm. It is possible to specify such things as upper bounds, lower bounds and insertion orders.

Once the external module has been implemented it must be compiled to create an executable module binary. If written in C++, the external module can be compiled by adapting sample Makefiles included in the LiSA installation. When the LiSA software is next run, the algorithm will be available for use on the problems that have been specified it can solve.

Another standard practice in developing external algorithms for LiSA is the creation of a help file. This file should be written in HTML so that online help within LiSA is available.

## **2.7 Possible Project Enhancements**

Upon completion of the minimum requirements (see 1.3) it would be possible to enhance the project in a number of ways.

One such enhancement would be the modification of the LTRPOM rule used in the software component. When the computational experiments have been completed it is likely that there will be certain instances of the open-shop problem for which the performance of the LTRPOM rule is unsatisfactory, for example, the production of schedules that are far from optimal. These instances could then be studied to

determine why the rule did not perform acceptably and subsequently the rule could be modified to perform better with such instances in the future.

Another enhancement would be the incorporation of the software component into the LiSA software itself. As the software component will be designed to use the modelling concepts of LiSA it would be feasible to adopt more LiSA conventions to facilitate external module development. This would result in the software component becoming a part of LiSA and being available for execution from within the LiSA GUI.

## **Chapter 3**

### **Algorithm Design**

#### **3.1 Choice of Platform and Language**

The choice of language comes down to a choice between Java, C and C++ due to their convenience. All three languages have development environments available on most platforms.

Java has good object orientation support and includes a large class library, which will accelerate the development of a software component. It also possesses the significant feature of platform independence. Java can have performance issues though due to it being interpreted on a virtual machine, opposed to being executed directly on the hardware.

C is generally acknowledged to be the best of these 3 languages for producing high performance code but it does not support object orientation.

C++ has good object orientation support, like java, and can produce high performance code, like C. As well as this, it usually produces code with considerably better performance than java.

Performance is an issue, as it is with most algorithms, but especially in this project due to possible large size of scheduling problems. Object orientation is also important when considering the supplied classes included with LiSA itself. The fact that these classes are written in C++ is also a major defining factor in language choice. For these reasons, C++ has been chosen as the development language.

The choice of platform comes down to a choice between Linux and Microsoft Windows 2000, as LiSA runs on both. Linux provides a stable and flexible development environment with the advantage of the operating system and its tools being free. Windows 2000 does have a number of stability and performance issues when in software development and the tools to develop Windows applications can cost money. For these reasons, Linux has been chosen as the development platform.

## **3.2 Algorithm Breakdown**

The LAPT/LTRPOM algorithms can be considered as a set of smaller algorithms. These functions can be considered separately and then pieced together to form a main algorithm using a certain set of data.

### **3.2.1 LiSA Objects**

When the software component is run it will first need to create a number of objects and input data into them. Input and output streams will also have to be created to retrieve this data and then output the final schedule when the main algorithm has completed.

Objects will be needed for the problem type, control parameters (although not required by the LAPT and LTRPOM rules) and the values themselves (processing times etc.). A schedule object is also needed to hold the final schedule when the algorithm has decided it.

Before the software component terminates the objects that have been created must all be deleted.

### **3.2.2 Set of Data**

First of all there must be two integers,  $m$  and  $n$ , to hold the number of machines and jobs respectively and a boolean value to hold the type of scheduling currently in use, whether by processing times or otherwise.

There must be a matrix holding the processing times that each job requires on each machine. This will be input when the algorithm starts and will not be changed during the algorithm.

There must be another matrix holding the availability of each job on each machine. This would indicate if a certain job has had processing on a certain machine and therefore signify which jobs and machines are still available.

A 'working' Latin rectangle must also be present so it can be filled in with job execution positions as the algorithm works. This Latin rectangle can then be used to output the schedule constructed by the algorithm.

A vector holding the next execution number on each machine can be used to fill in the 'working' Latin rectangle. When a job has been scheduled on a machine the 'working' Latin rectangle can be filled in with the appropriate position vector element relating to that machine. The position vector element can then be incremented for the next time it is required.

Another vector is required to hold data on the processed status on each job over every machine. This vector is similar to the availability matrix in that it indicates if a certain job has received processing, but the vector indicates if the job has received processing on any machine, opposed to a particular machine. This processed vector is used to indicate if a job has received processing so that it can be excluded from the job calculation part of the algorithm as it schedules "among the jobs that have not yet received processing" (Pinedo 2002:188).

### **3.2.3 Main Algorithm**

The main body of the algorithm must decide what type of scheduling should currently be used, then select a job number and a machine number, and then fill in the 'working' Latin rectangle, availability matrix and processed vector and increment the appropriate element in the position vector.

To determine what type of scheduling should currently be used a 'calculate schedule' function is needed. This would decide whether to schedule by processing times if some jobs have received no processing on any machine or otherwise if all jobs have received processing on any machine. To determine the machine to schedule on and the job to schedule on it a 'calculate machine' function and a 'calculate job' function are needed. Once this has been ascertained the 'working' Latin rectangle, availability matrix, processed vector and position vector can all be updated.

All this must be done while there is at least one job to be scheduled on one machine, which can be calculated with a 'free' function. The algorithm can then output the schedule from the 'working' Latin Rectangle and terminate.

#### **3.2.3.1 Free**

This function can cycle through each machine, checking if it available to schedule on by running a 'machine available' function. If every machine is not available then the algorithm has finished. If at least one machine is available then the algorithm has not finished and can continue.

#### **3.2.3.2 Calculate Schedule**

This function can cycle through each job looking at the processed vector as it goes. If at least one job has not received any processing on any machine then the algorithm will schedule by processing times. If all jobs have received processing on any machine then the algorithm will schedule jobs in an arbitrary order.

#### **3.2.3.3 Machine Available**

This function can cycle through each job on a given machine looking at the availability matrix as it goes. If every job is not available then the machine is not available for scheduling on. If at least one job is available then the machine is available for scheduling on.

#### **3.2.3.4 Calculate Machine**

Whenever a machine is freed it should be selected as the next machine to be scheduled on, assuming it has not already executed all jobs on it. A machine's availability can be checked with the 'machine available' function, which is already used in the 'free' function. A machine's 'free-ness' can be calculated with a 'machine



sum' function to find the machine with the least processing on it so far. When a suitable machine has been selected the machine number can be returned to the main algorithm.

#### **3.2.3.5 Machine Sum**

This function can cycle through each job on a given machine looking at the availability and processing times matrices as it goes. The sum of processing times of the jobs that are not available will give the current processing time on that machine. The 'calculate machine' function can then determine the machine with the least processing on it.

#### **3.2.3.6 Calculate Job**

Once a machine has been selected a job must be selected to be processed on that machine. If the algorithm is scheduling by processing times then each available job on the machine in question has the sum of it's processing time on the other machine(s) calculated by a 'job sum' function. These times are then compared to find the job with the largest time, which will give the job number to be returned to the main algorithm. If the algorithm is not scheduling by processing times then a job is arbitrarily chosen from all available jobs on the machine in question.

#### **3.2.3.7 Job Sum**

This function can cycle through each machine looking at a given job and the availability matrix as it goes. The sum of processing times of each available job on each machine is calculated and then the processing time of the given job on the previously determined machine is subtracted. This gives the sum of the given job's processing times on the other machine(s).

### **3.3 Evaluation Criteria**

The software component will need to be evaluated once implemented. A number of evaluation criteria can therefore be specified:

11. The choice of language and platform is appropriate.
12. The software component solves the open-shop problem using Pinedo's rules and the modelling concepts of LiSA.
13. The software component executes correctly.
14. The software component is simple to use and documentation is available.
15. The software component produces optimal schedules for the open-shop problem with two machines and the makespan as the objective function,  $O2 \parallel C_{\max}$ .
16. The software component produces schedules for the open-shop problem with more than two machines and the makespan as the objective function,  $Om \parallel C_{\max}$  with  $m \geq 3$ .
17. The software component is written in a way that will facilitate future development.

## **Chapter 4**

### **Algorithm Implementation**

#### **4.1 Input and Output**

The software component takes input and gives output via 2 files specified in the command line interface. This interface works both from with the LiSA GUI and independently from a command line in Linux.

The software component takes two arguments from the command line: the first being the input file and the second being the output file. It then uses these files to create an input and output stream:

```
ifstream i_strm(argv[1]);  
ofstream o_strm(argv[2]);
```

The data from the input stream is then input into certain LiSA objects (see 4.2). The resulting schedule from the algorithm is output using the output stream.

#### **4.2 LiSA Objects**

Four objects need to be created at the start of the software component. These will hold the problem type, the control parameters (although not required by the LAPT and LTRPOM rules), the values themselves (processing times etc.) and the final schedule when the algorithm has decided it. These objects are created using the `new` keyword:

```
Lisa_ProblemType * lpr = new Lisa_ProblemType;  
Lisa_ControlParameters * sp = new Lisa_ControlParameters;  
Lisa_Values * my_values = new Lisa_Values;  
Lisa_Schedule * out_schedule = new Lisa_Schedule(n, m);
```

The data to be input into the first three objects comes from the specified input stream (see 4.1) and must be in the following order:

```

i_strm >> (*lpr);
i_strm >> (*sp);
i_strm >> (*my_values);

```

The data to be input into the last object comes from the 'working' Latin rectangle (see 4.3). This object is then used to output the schedule onto the output stream:

```

o_strm << * out_schedule;

```

Before the software component terminates the objects that have been created must all be deleted using the `delete` keyword:

```

delete out_schedule;
delete my_values;
delete sp;
delete lpr;

```

### **4.3 Data Structures**

The two integers, `m` and `n`, used to hold the number of machines and jobs respectively, can be input from the `Lisa_Values` object by using the `get_m()` and `get_n()` methods. The type of scheduling currently in use is held in the boolean value `PTSchedule`.

The processing times matrix, `T`, is of type `Lisa_Matrix<double>` and size `n x m`. The data can be input from the `Lisa_Values` object. The availability matrix, `A`, and the 'working' Latin rectangle, `PR`, are both of type `Lisa_Matrix<int>` and size `n x m`.

The position vector, `position`, is of type `Lisa_Vector<int>` and size `m`. All of its elements are first set to 1 so that they can be incremented correctly at a later stage.

The processed vector, `processed`, is of type `Lisa_Vector<int>` and size `n`. All of its elements are first set to 0 to indicate that no jobs have yet received any processing time on any machine.

## **4.4 Main Algorithm**

The main body of the algorithm must first decide what type of scheduling should currently be used by calling the `calculateSchedule()` function and storing the result in the boolean value `PTSchedule`.

The machine to schedule on must be decided next by calling the `calculateMachine()` function and storing the result in the integer `machine`. Now the job to schedule on the chosen machine must be decided by calling the `calculateJob()` function and storing the result in the integer `job`.

Once the machine and job numbers have been ascertained the 'working' Latin rectangle, availability matrix, processed vector and position vector can all be updated.

This process must continue while there is at least one job to be scheduled on one machine, which can be calculated with the function `free()`. The main body of the algorithm is therefore encapsulated in a while loop.

### **4.4.1 free()**

This function is of boolean type. It takes the availability matrix, the number of machines and the number of jobs as parameters. It uses a for loop to go through each machine and check if the machine is available by calling the `machineAvailable()` function. If the sum of unavailable machines is equal to the total number of machines, the function returns false, otherwise it returns true.

### **4.4.2 calculateSchedule()**

This function is of boolean type. It takes the processed vector and the number of jobs as parameters. It uses a for loop to go through each job and check if the job has received any processing on any machine. If the sum of processed jobs is equal to the total number of jobs, the function returns false, otherwise it returns true, as some jobs have not received any processing on any machine yet.

#### **4.4.3 machineAvailable()**

This function is of boolean type. It takes the availability matrix, a particular machine number and the number of jobs as parameters. It uses a for loop to go through each job and check if the job has been processed on the given machine. If the sum of jobs processed on the given machine is equal to the total number of jobs, the function returns false, otherwise it returns true.

#### **4.4.4 calculateMachine()**

This function is of type integer. It takes the availability matrix, the processing times matrix, the number of machines and the number of jobs as parameters. It uses a for loop to go through each machine; then an if statement to check if the machine is available by calling the `machineAvailable()` function and then another if statement to check if this machine has a lesser processing time on it so far compared to the current machine by calling the `machineSum()` function for both machines. This finds the next available machine and the function returns the appropriate machine number.

#### **4.4.5 machineSum()**

This function is of type double. It takes the availability matrix, the processing times matrix, a particular machine number and the number of jobs as parameters. It uses a for loop to go through each job and check if the job has been processed on the given machine. The sum of processing times of the jobs that are not available will give the current processing time on that machine, which is the value returned by the function.

#### **4.4.6 calculateJob()**

This function is of type integer. It takes the availability matrix, the processing times matrix, the processed vector, a particular machine number, the number of machines and the number of jobs as parameters. If the algorithm is scheduling by processing

times it uses a for loop to go through each job; then an if statement to check if the job is available on the given machine and the job has not received any processing on any machine and then another if statement to check if this job has a greater processing time on the other machine(s) compared to the current job by calling the `jobSum()` function for both jobs. If the algorithm is not scheduling by processing times it uses a for loop to go through each job and check if the job is available on the given machine. It then selects an available job arbitrarily. This finds the next job to schedule on the given machine and the function then returns the appropriate job number.

#### **4.4.7 jobSum()**

This function is of type double. It takes the availability matrix, the processing times matrix, a particular machine number, the number of machines and a particular job number as parameters. It uses a for loop to go through each machine and check if the given job has been processed on that machine. The sum of processing times of the jobs that are available minus the processing time of the given job on the given machine will give the sum of the given job's processing times on the other machine(s), which is the value returned by the function.

### **4.5 Compilation and Installation**

Makefiles for other external modules are included in the LiSA installation. These provide a basis for creating a Makefile for the software component.

Assuming the external module is written without errors, executing “`make depend`”, “`make compile`” and “`make install`” commands will create an executable module binary and install it in the correct directory. When the LiSA software is next run, the algorithm will be available for use on the problems that have been specified it can solve.

## **4.6 Testing**

All parts of the software component were tested as they were written and the main algorithm was tested each time it was built upon. This involved inserting lines of code to output the current status of functions and variables and then executing the module on the command line using sample input files. This is a simple way to pace through the algorithm and check that it is behaving as expected.

A number of issues, including two major ones, were detected using this method and subsequently fixed (see 5.4 and 5.5).



## **Chapter 5**

### **Implementation Issues**

#### **5.1 Introduction**

As with all software development, there have been pauses in the advancement of the software component due to programming issues and unforeseen problems. This chapter details a number of major issues that arose, in chronological order.

#### **5.2 LiSA Compilation**

I installed a copy of the LiSA source code for Linux into my home directory on a School of Computing Linux workstation, which was successful. I then attempted to compile the source code but was faced with a multitude of errors. After some constructive communications via email with Graham Hardman (School of Computing Support) we obtained a working copy of LiSA through the editing of a number of key files. This was all well and good but I did not want to create an external module for LiSA that only worked with my edited version.

It seemed obvious that the problem related to the compiler I was/wasn't using so I contacted a LiSA developer, Marc Mörig, via email. I explained I was using tcl version 8.3 and g++ version 3.1.1 and Marc confirmed that LiSA did not like gcc 3, but that this had changed for the new version still under development.

If I could use an older compiler then I would be able to develop an external module using the same basic code as LiSA itself. This compiler was to be found on a particular workstation in the School of Computing, which held the School's previous compiler, gcc 2.96. By remotely logging into this workstation and setting a path to the old compiler I could compile LiSA and my external algorithm:

```
setenv PATH /usr/local/kgcc/bin:$PATH
```

### **5.3 German**

As LiSA was first developed at Otto-von-Guericke University Magdeburg, Germany, a large amount of comments and explanations are in German. I do not speak very good German, especially technical words as would be used in this environment. This made it very difficult to grasp the basics of external module development in LiSA. I could pick my way through lines of code in other external modules but when statements are not self-descriptive and their explanations are in German it became a painstaking process to understand how these modules operated.

I attempted to translate most of the words and statements I could not understand by using online translators available from the World Wide Web, but these proved almost useless, bringing up words that were too general and statements that made no sense.

Included with the LiSA installation were some HTML files giving help on incorporating external algorithms, which were written in English. These help files were far too basic though, simply glossing over the specifics of module integration and development. A developer's manual was also included, but this was merely a short specification of classes, functions and macros, which, while useful, did not clarify what was required for incorporating external modules.

Eventually I came across a paper entitled "LiSA Module Concept", by Dornheim (2003), which detailed what was required to create an external module for use in LiSA. I was able to use this to begin work on the external module.

### **5.4 Select First Current Machine**

It became apparent during the development of the software component that the `calculateMachine()` function was not operating correctly. At some points it would keep selecting machine 0, the first machine, even though it was not available. It became clear that the function needed to arbitrarily select an available machine to use in its `machineSum()` comparisons. Currently it was just selecting machine 0 at the start of the function without checking whether it was available or not.

A for loop was added to go through each machine and then an if statement to check if the machine was available by calling the `machineAvailable()` function. This selected an available machine arbitrarily and fixed the problem.

### **5.5 Select First Current Job**

Similarly to the first current machine problem, it became apparent that the `calculateJob()` function was not operating correctly. At some points it would keep selecting job 0, the first job, even though it was not available on the given machine. It became clear that the function needed to arbitrarily select an available job on the given machine to use in its `jobSum()` comparisons. Currently it was just selecting job 0 at the start of the function without checking whether it was available or not.

A new function of type integer, `selectCurrent()`, was created. It was to take the availability matrix, the processed vector, a particular machine number and the number of jobs as parameters. A for loop was added to go through each job and then an if statement to check if the job was available on the given machine and the job had not received any processing on any machine. This selected an available job on the current machine arbitrarily and returned it to the `calculateJob()` function, fixing the problem.

## **Chapter 6**

### **User Guide**

This chapter contains an approximation of the user guide created for the software component, slightly edited for this report. The actual user guide was submitted with the software component.

Note: This is not a user guide for LiSA itself.

#### **6.1 Introduction**

The software component contains an algorithm to solve the open-shop problem with an arbitrary number of machines and the makespan as the objective function,  $Om \parallel C_{\max}$  with  $m \geq 2$ . It is based on LiSA's modelling concepts and Pinedo's LAPT and LTRPOM rules. The LAPT rule is an exact algorithm for the open-shop problem with two machines and the makespan as the objective function,  $O2 \parallel C_{\max}$ . The LTRPOM rule is a construction heuristic for the open-shop problem with more than two machines and the makespan as the objective function,  $Om \parallel C_{\max}$  with  $m \geq 3$ . This means that it produces an optimal schedule for the problem  $O2 \parallel C_{\max}$ , but cannot guarantee an optimal schedule for the problem  $Om \parallel C_{\max}$  with  $m \geq 3$ , as this problem is NP-complete.

#### **6.2 Compilation and Installation**

The software component comes with a set of Makefiles, algorithm description files and HTML help files. These files must be placed in the 'lapt' directory in `${LISAHOME}/source/external/`.

Once this is done you can compile and install the module. While in the 'lapt' directory, executing "make depend", "make compile" and "make install" commands will create an executable module binary and install it in the correct

directory,  $\${LISAHOME}/bin/$ . The algorithm description files and HTML help files will also be installed into the correct directories:

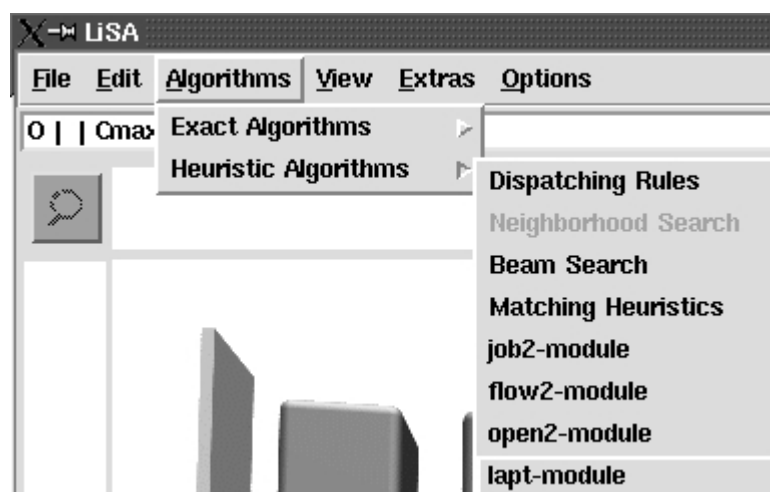
```
cslin200% make compile
cslin200% make install
cp ../../../../compiled/lapt ../../../../bin/lapt
cp lapt_engl.alg ../../../../data/alg_desc/language/english/lapt.alg
cp lapt_ger.alg ../../../../data/alg_desc/language/german/lapt.alg
cp lapt_geru.alg ../../../../data/alg_desc/language/german_u/lapt.alg
cp lapt_engl.html ../../../../doc/language/english/lapt.html
cp lapt_ger.html ../../../../doc/language/german/lapt.html
cslin200%
```

**Figure 6.1 – Command Line Compilation and Installation of the Software Component**

When the LiSA software is next run, the algorithm will be available for use on the problems that have been specified it can solve.

### **6.3 Executing from within LiSA**

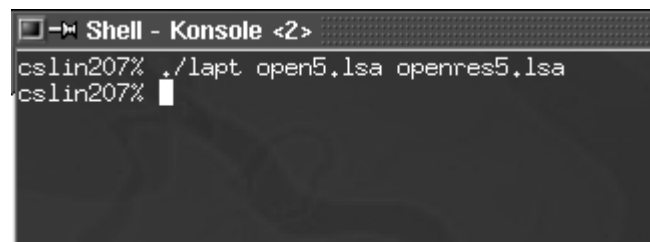
Once you have specified your problem type and entered your problem parameters in LiSA you will be available to select an algorithm to solve your problem. If you have selected an open-shop problem with no constraints and the makespan as the objective function you will be able to use this software component. To do this you must select 'lapt-module' from the 'Heuristic Algorithms' sub-menu in the main 'Algorithms' menu. Once the algorithm is complete, a Gantt chart of the constructed schedule will be displayed.



**Figure 6.2 – Algorithm Menu in LiSA Showing 'lapt-module'**

## **6.4 Executing from the Command Line**

You can also execute the software component from the Linux command line. To do this you must first enter the directory that holds the executable module binary, `${LISAHOME}/bin/`. You can then run the software component by specifying two files and typing `${bin}./lapt input.lsa output.lsa`. For example:



```
Shell - Konsole <2>
cslin207% ./lapt open5.lsa openres5.lsa
cslin207%
```

**Figure 6.3 – Command Line Execution of Software Component**

The input file has to be an LSA file, which is a LiSA data file. The output file will be created in the same format. Once the algorithm is complete you will be returned to the command line. The output file, with the name you specified, contains the constructed schedule in the form of a Latin rectangle. Examples of an input and output file can be found in Appendix C.

## **Chapter 7**

### **Computational Experiments**

#### **7.1 Introduction**

To analyse the performance of the software component against the known algorithms from LiSA a set of instances of open-shop problems using random number generation needed to be created. The software component and the known algorithms then solved these instances and the values of their objective functions, in this case the makespans, were noted.

#### **7.2 Open-Shop Instances**

Problems consisting of 2 machines, 5 machines and 10 machines were used. Each problem was then divided into 10 jobs, 20 jobs, 50 jobs or 100 jobs. Each one of these was then executed three times, each time with a different randomly generated data set. This produced 36 instances of differing open-shop problems.

#### **7.3 Algorithms Used**

As well as using the software component (SC), the known algorithms used from LiSA were:

Exact	Branch and Bound (BB)	
Heuristic	Dispatching Rules	Longest Processing Time (LPT)
		Shortest Processing Time (SPT)
		Random (RAND)
	Neighbourhood Search	Simulated Annealing (SA)
		Threshold Accepting (TA)
		Tabu Search (TS)
		Iterative Improvement (II)

The neighbourhood search algorithms are improvement heuristics and so need to start with a schedule before trying to find a better 'similar' schedule. The random dispatching rule was therefore used to construct an initial schedule.

The branch and bound algorithm can take a long time to terminate. For these experiments, if it had not terminated after an hour, the current best scheduled it had produced was used.

## **7.4 Results**

Below is a table of the resulting makespans obtained from each algorithm:

m	n	SC	LPT	SPT	RAND	SA	TA	TS	II	BB
2	10	208	208	208	208	208	208	208	208	208
		252	252	252	252	252	252	252	252	252
		306	306	306	306	306	306	306	306	306
	20	509	509	509	509	509	509	509	509	511
		593	593	593	593	593	593	593	593	593
		512	512	512	512	512	512	512	512	512
	50	1349	1349	1349	1349	1349	1349	1349	1349	1349
		1304	1304	1304	1304	1304	1304	1304	1304	1304
		1339	1339	1339	1339	1339	1339	1339	1339	1339
	100	2647	2647	2647	2647	2647	2647	2647	2647	2647
		2569	2569	2569	2569	2569	2569	2569	2569	2569
		2754	2754	2754	2754	2754	2754	2754	2754	2754
5	10	334	245	264	265	247	264	258	262	245
		338	286	286	312	286	286	286	286	286
		407	325	328	322	328	328	328	328	322
	20	602	562	562	562	562	562	562	562	562
		633	521	521	541	542	547	521	521	521
		677	546	546	558	546	558	546	546	546
	50	1892	1478	1478	1478	1478	1478	1478	1478	1663
		1655	1393	1393	1393	1393	1393	1393	1393	1507
		1562	1309	1309	1309	1309	1309	1309	1309	1487
	100	2927	2694	2694	2694	2694	2694	2694	2694	2964
		2967	2683	2683	2683	2683	2683	2683	2683	2893
		2987	2560	2560	2560	2560	2560	2560	2560	2774
10	10	516	324	336	341	341	341	326	335	324
		683	322	332	338	338	338	325	332	322
		722	335	335	337	337	337	325	332	325
	20	839	545	545	549	549	549	545	549	673
		954	703	703	703	703	703	703	703	823
		863	615	683	615	615	615	615	615	717
	50	2137	1552	1552	1552	1552	1552	1552	1552	2207
		1904	1386	1386	1386	1386	1386	1386	1386	1983
		2084	1446	1446	1446	1446	1446	1446	1446	2066
	100	3279	2798	2798	2798	2798	2798	2798	2798	3616
		3249	2680	2680	2680	2680	2680	2680	2680	3478
		3323	2785	2785	2785	2785	2785	2785	2785	3502



## **Chapter 8**

### **Evaluation**

#### **8.1 Minimum Requirements**

The minimum requirements for the project were specified in the Project Overview (see 1.3) and are reproduced here:

4. To implement Pinedo's rules as a software component using the modelling concepts of LiSA.
5. To learn and analyse the known algorithms for the open-shop problem.
6. To perform computational experiments to analyse the performance of the software component against the known algorithms from the library of scheduling algorithms LiSA.

The software component produced has not only achieved these minimum requirements, it has also exceeded them by embracing one of the suggested project enhancements. The incorporation of the software component into the LiSA software itself has also been achieved.

#### **8.2 Software Component**

I can evaluate the software component by applying the evaluation criteria specified in Algorithm Design (see 3.3):

- The choice of language and platform is appropriate.

C++ was a good choice of language as it had good object orientation support and produced high performance code. It also assisted in the integration of the software component with LiSA, as the supplied classes included with LiSA are written in C++. Linux was a good choice of platform as it was very stable and although a few problems arose with compilation, they were relatively easily solved.

- The software component solves the open-shop problem using Pinedo's rules and the modelling concepts of LiSA.

The software component was created using Pinedo's rules and the modelling concepts of LiSA and did manage to solve all open-shop problems tested on it.

- The software component executes correctly.

The software component had no problems with command line execution, as long as it was given a valid input file. If the input file were invalid an error would occur.

- The software component is simple to use and documentation is available.

The software component is very simple to use as it can only be used in two ways, from the command line with two arguments or from the within the LiSA GUI. A user guide is also available containing a step-by-step guide on using the software component.

- The software component produces optimal schedules for the open-shop problem with two machines and the makespan as the objective function,  $O2 || C_{max}$ .

For all two machine open-shop problems used in the computational experiments (see 7), the software component produced optimal schedules.

- The software component produces schedules for the open-shop problem with more than two machines and the makespan as the objective function,  $Om || C_{max}$  with  $m \geq 3$ .

The software component does manage to produce schedules for all open-shop problems given to it, but frequently, for problems with more than 2 machines, the output schedules are far from optimal (see 7).

- The software component is written in a way that will facilitate future development.

The software component has been designed to be as expandable as possible. The use of common functions and sensible variable names provide an easy starting point to understand the workings of the software component. The extensive comments found throughout the programme code also aid understanding.

Due to the software component embracing one of the suggested project enhancements it is possible to specify one more evaluation criterion:

- The software component executes correctly from within the LiSA GUI.

The software component had no problems with LiSA GUI execution at all, partly due to the fact that you cannot enter invalid data values into LiSA.

### **8.3 Project Schedule**

The project schedule was specified in the Project Overview (see 1.5). The project schedule was followed very well up to the “Make Software” point, where the some major issues became apparent (see 5). Although not too serious, these issues did eat a lot of time out the schedule, causing the remaining tasks to have to be done at a more rapid pace.

## **Chapter 9**

### **Conclusions and Future Work**

#### **9.1 As It Is**

The capabilities of the software component are good. It works correctly and produces schedules as it was expected to. It can be executed from the command line or from within the LiSA GUI in Linux. It produces optimal schedules for the open-shop problem with 2 machines and the makespan as the objective function. However, for problems with more than 2 machines, the output schedules are far from optimal. This was somewhat to be expected, as there was no guarantee that the LTRPOM rule could produce near optimal schedules on a regular basis.

The efficiency of the software component is excellent when considering the size of problem it is solving. It executes very quickly both from the command line and from within LiSA. It is not particularly resource-hungry and no slow-down is experienced when running the software component and other programmes concurrently on the same workstation.

#### **9.2 As It Could Be**

There are a number of future developments that could be applied to the software component to enhance its capabilities.

One such development could be the modification of the LTRPOM rule as mentioned in Possible Project Enhancements (see 2.7). There were certain instances of the open-shop problem for which the performance of the LTRPOM rule was unsatisfactory and the software component produced schedules that were far from optimal. These instances could be studied further to determine why the rule did not perform acceptably and subsequently the rule could be modified to perform better with such instances in the future.

Another addition could be the potential to accept different processing characteristics and constraints because currently the software component will not accept any. Many

different constraints exist that could be incorporated into the software component to give it a larger range of problems to solve. These include release dates, due dates, preemptions, and weights.

## **References**

Bräsel, H. ***Latin Rectangles in Scheduling Theory – a Basic Modelling Concept of LiSA***. Magdeburg: Otto-von-Guericke University Magdeburg, 1996.

Dornheim, L. ***LiSA Module Concept***. Magdeburg: Otto-von-Guericke University Magdeburg, 2003.

Gonzalez, T and Sahni, S. ***Open Shop Scheduling to Minimize Finish Time***. Minneapolis: University of Minnesota, 1976.

*LiSA-Homepage* [Online]. 2003 [Accessed 10<sup>th</sup> April 2003]. Available from World Wide Web: <<http://fma2.math.uni-magdeburg.de/~lisa/>>

Pinedo, M. ***Scheduling: Theory, Algorithms, and Systems***, 2<sup>nd</sup> Edition. New Jersey: Prentice Hall, 2002.

# **Appendix A**

## **Project Reflection**

This project did not require an extremely large amount of research nor an extremely large amount of programming, but it did require an extremely large amount of thinking on your own. This, or any similar project, had not been attempted before in the School of Computing and it was not a widely discussed subject outside of the University either so I felt that a lot of innovation was required.

Overall though, I did enjoy the project and I am glad that I chose this particular field to study in. I have learnt a lot about scheduling theory and how it can be applied in the real world. I have also improved my programming skills and I am now a lot more confident with C++ compared to before I started this project.

The one major lesson that I learnt from this project is the importance of time management. This includes constructing a schedule, trying to keep to that schedule and the worth of a back-up plan if the schedule fails. I had a number of problems when implementing the software component and this resulted in a major rush to finish the report writing close to the project deadline. I should have initially made the project schedule shorter to compensate for problems near the end and written a back-up plan stating what to do if a major disruption occurs. Of course I could not foresee the problems ahead but more careful planning could have reduced the effect of these problems.

Another lesson that I learnt is that you can't trust computers. They seem to be doing what you want them to do but then all of a sudden they do something completely unexpected. I suppose that in the sphere of programming, if the computer is not behaving as you would like, it is actually your own fault for not coding correctly. So in actual fact I have learnt that you can't trust yourself! Basically, this boils down to the extreme importance of testing your software to make 100% sure it works as you wish.

If I were to undertake a similar project in the future I would start a lot earlier and plan to finish a lot sooner so as to budget for problems during certain project phases.

## **Appendix B**

### **Example ALG Files**

#### **Example 1**

```
<GENERAL>
Name= Branch & Bound
Type= constructive
Call= bb
Code= external
Help= bbwindow.htm
</GENERAL>

<EXACT>
<PROBLEMTYPE>
Lisa_ProblemType= { O / r_i / SumWiTi }
</PROBLEMTYPE>
<PROBLEMTYPE>
Lisa_ProblemType= { J / r_i / SumWiTi }
</PROBLEMTYPE>
<PROBLEMTYPE>
Lisa_ProblemType= { O / r_i / SumWiUi }
</PROBLEMTYPE>
<PROBLEMTYPE>
Lisa_ProblemType= { J / r_i / SumWiUi }
</PROBLEMTYPE>
</EXACT>

<HEURISTIC>
</HEURISTIC>

<PARAMETERS>
long NB_SOLUTIONS 1 "Number of solutions"
double LOWER_BOUND 0 "Lower bound"
double UPPER_BOUND 1000000 "Upper bound"
string INS_ORDER ( RANDOM LPT ) "Insertion order"
string BOUNDING ( EXTENDED NORMAL ) "Bounding"
</PARAMETERS>
```



## **Example 2**

```
<GENERAL>
Name= open2-module
Type= constructive
Call= open2
Code= external
Help= open2.html
</GENERAL>

<EXACT>
<PROBLEMTYPE>
Lisa_ProblemType= { 0 / p_ij=1 / Cmax }
</PROBLEMTYPE>
</EXACT>

<HEURISTIC>
<PROBLEMTYPE>
Lisa_ProblemType= { 0 / / Cmax }
</PROBLEMTYPE>
</HEURISTIC>

<PARAMETERS>
string NAME ( was das soll ) "name"
double UPPER_BOUND 1000000 "Obere Schranke"
long NB_STEPS 1 "Anzahl der Schritte"
</PARAMETERS>
```

## **Appendix C**

### **Example LSA Files**

#### **Example 1 – Input File**

```
<PROBLEMTYPE>
Lisa_ProblemType= { 0 / / Cmax }
</PROBLEMTYPE>

<CONTROLPARAMETERS>
</CONTROLPARAMETERS>

<VALUES>
m= 2
n= 8
PT= {
  { 35 10 }
  { 10 5 }
  { 21 4 }
  { 1 38 }
  { 31 20 }
  { 19 25 }
  { 14 21 }
  { 47 10 }
}

SIJ= {
  { 1 1 }
  { 1 1 }
  { 1 1 }
  { 1 1 }
  { 1 1 }
  { 1 1 }
  { 1 1 }
  { 1 1 }
}

</VALUES>
```

## **Example 2 – Output File**

<SCHEDULE>

m= 2

n= 10

semiactive= 1

LR= {

{ 6 7 }

{ 7 8 }

{ 8 9 }

{ 1 2 }

{ 10 1 }

{ 2 3 }

{ 3 4 }

{ 9 10 }

{ 4 5 }

{ 5 6 }

}

CIJ= {

{ 88 189 }

{ 98 194 }

{ 119 198 }

{ 1 58 }

{ 197 20 }

{ 20 83 }

{ 34 104 }

{ 166 208 }

{ 43 153 }

{ 53 179 }

}

</SCHEDULE>