1-1-2006

Heuristics for batching jobs under weighted average completion time

Lewis A Raymond
University of Nevada, Las Vegas

HEURISTICS FOR BATCHING JOBS UNDER WEIGHTED AVERAGE COMPLETION TIME

by

Lewis A. Raymond

Bachelor of Science
University of Nevada, Las Vegas
2003

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2006

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

MAY 3 , 2006

The Thesis prepared by

LEWIS A. RAYMOND

Entitled

HEURISTICS FOR BATCHING JOBS UNDER WEIGHTED AVERAGE COMPLETION TIME.

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Heuristics for Batching Jobs Under Weighted Average Completion Time

by

Lewis A. Raymond

Dr. Wolfgang Bein, Examination Committee Chair
Associate Professor of Computer Science
University of Nevada, Las Vegas

Batching problems are machine scheduling problems, where a set of jobs with given processing requirements has to be scheduled on a single machine. The set of jobs has to be partitioned into subsets to form a sequence of batches. A batch combines jobs to run jointly, and each job's completion time is defined to be the completion time of the entire batch. For a batching problem, it is also assumed that when each batch is scheduled, it requires a setup time. One seeks to find a schedule that minimizes the total weighted completion time.

This problem is NP-complete, but the problem can be solved efficiently in $O(n \log(n))$ time if the order of the jobs is given. This is accomplished through a non-trivial reduction to on-line matrix searching in a totally monotone array. An implementation of this algorithm is part of the thesis work.

To remove the requirement of a fixed order and thus to solve the original NP-complete batching problem, the space of permutations is searched using a genetic algorithm technique. The implementation uses a library of object-oriented functions, GAlib, to implement genetic algorithms. This highly versatile library

was written by Mathew Wall of MIT.

The thesis also seeks to find techniques to obtain an upper bound, which can be used to measure the quality of the solutions found by the heuristic.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost I would like to thank Dr. Bein for his paramount guidance through this compelling project. I would like to remark about his tendency to allow me this thesis opportunity given the exceptional circumstances. I would also like to remark about his patient mannerism during our meetings; in short, he made this project seem possible for me.

I would like to thank Dr. Minor, Dr. Gewali, and Dr. Selvaraj for their participation as members of my advisory committee. I enjoyed their classes during my undergraduate, and graduate career at UNLV, and their lectures and counseling were very beneficial.

I would like to thank Dr. Larmore for his direct and indirect input of solutions to many problems that arose during the course of this project.

I would like to thank my mentor Lee Misch. She has provided invaluable advise and support during my entire college experience at UNLV. Lee Misch always lent an ear to listen to me unload frustrations and victories.

Even though they are not affiliated with this thesis project, I would like to thank Dr. Taghva and Dr. Nartker, the directors of the Information Science Research Institute. Thank you for giving me the opportunity to attend graduate school. I am very grateful for their support during my graduate course work, and the experience I gained as a research assistant.

Lastly, I would like to thank my family for allowing me to ignore them while I

feverishly worked on this project. Thanks to my wife Janice for her unconditional love and support during this endeavor. Thanks to my daughter Tay'Lor for understanding the monster within as I released my frustrations in her direction in the form of lunatic rantings. She displayed maturity and responsibility during my absence.

Thank you all.

CHAPTER 1

INTRODUCTION

Batching problems are machine scheduling problems, where a set of jobs with given processing requirements has to be scheduled on a single machine. The set of jobs has to be partitioned into subsets to form a sequence of batches. An s-batching probiem is specified as the jobs of a batch are processed in series. On the other hand, if one wishes to use multiple machines, the jobs of each batch are processed in paraiiei. This is known as the p-batching problem, and its complexity is currently still open. In this thesis, the s-batching probiem is tackled.

The primary part of this thesis is to describe heuristics and approximations for the s-batch problem under a weighted average completion time. Since finding an optimal solution to this problem is NP-compiete, a genetic algorithm is used to find good solutions in the search space. Experiments are performed in which the genetic algorithm reports such solutions.

A lower bound is proved and a 2-approximation aigorithms is given. The results from the genetic algorithm are then compared to these bounds. Furthermore another approximation aigorithm is given, for which the conjecture is that it will be a better than two approximation.

Central to the scoring within the genetic aigorithm is the fact that in the case where the order of the jobs is given, the batching problem can be solved in time

complexity of $O(n \log(n))$ using a efficient searching technique to find row minima in a totally monotone matrix.

The layout of this thesis paper is as follows. Chapter 2 gives a formal definition of the s-batch problem. It describes what weighted average completion time means, and it shows that the s-batching problem is NP-complete. Chapter 3 explains how, given an instance of jobs, an approximation algorithm can compute a solution that is no worse that twice the optimal solution. This approximation algorithm is used to compute an upper and a lower bound. Also another approximation algorithm is given. Chapter 4 illustrates monotone matrix searching, which is central to the case of s-batching with fixed order. It is shown that a Monge matrix implies the matrix to be totally monotone. Further monotone matrix searching under an on-line matrix protocol is given. This protocol is needed for the application of matrix searching to dynamic programming in Chapter 5, where is is shown that a shortest path problem, using a directed acyclic graph, can be reduced to the s-batching problem. Chapter 6 then defines the genetic algorithm. Implementation issues relating to GAlib, a C++ library, are also given. Finally, Chapter 7 summarizes the experiments performed, and gives results.

Background

Tables 1.1 and 1.2 show various batching problems. The problems are given using the notation $\alpha \mid \beta \mid \gamma$. The letter α expresses the number of machines, which for these problems the number of machines is always one. The letter β expresses any restriction placed on the set of jobs. The letter γ expresses the

objective function (Albers & Brucker, 1993).

Precedence Constraints (prec): The precedence constraints allows certain jobs to be processed before other certain jobs can start being processed in the schedule. Precedence constraints are represented with a directed acyciic graph in which each job is represented by a node of the graph. Job J_i comes before job J_j if there is a directed edge from i to j (Brucker, 2004).

Series Batch (s-batch): The jobs of each batch in the schedule are processed one after the other, such that job if J_i is before job J_j in the batch, then job J_i completes before J_j starts (Brucker, 2004).

Chains (chains): The chains of a problem are part of the precedence constraints. If the directed acyclic graph representation of the precedence constraints has jobs nodes where each node has at most one predecessor and at most one successor, then the constraints are referred to as chains (Brucker, 2004).

Processing Time (p_i) : Job J_i is processed on one machine, and requires time p_i. If p_i is present, then J_i is restricted to taking oniy as much time as allowed. For example, if $p_i=p$ then J_i is only allowed a time unit of p, or if $p_i=1$ then J_i is oniy allowed 1 time unit. Otherwise if p_i is not present, then J_i time is not restricted (Brucker, 2004).

Release Time (r_i) : Job J_i arrives at the system at time r_i, and that time is the earliest it may be processed. If r_i is not present, the J_i may start at any time (Brucker, 2004).

Maximum Lateness (L_{max}) : Of the set of jobs that are late $(L_1,\ldots,L_n)$

maximum lateness is $max(L_1,\ldots,L_n)$ (Brucker, 2004).

$(\sum C_i)$: denotes the total unweighted completion time (Brucker, 2004).

$(\sum w_i C_i)$: denotes the total weighted completion time (Brucker, 2004).

$(\sum U_i)$: denotes the total unweighted number of jobs that are tardy (Brucker, 2004).

$(\sum w_i U_i)$: denotes the total weighted number of jobs that are tardy (Brucker, 2004).

$(\sum T_i)$: denotes the total unweighted tardiness (Brucker, 2004).

Table 1.1: Polynomial Solvable s-Batch Problems (Brucker, 2004)

Problem	Time Complexity	Reference
1 | prec; s-batch | L_{max}	$O(n^2)$	Ng, Cheng, & Yaun, 2002
1 | prec; $p_i = p$; s-batch | $\sum C_i$	$O(n^2)$	Albers & Brucker, 1993
1 | s-batch | $\sum C_i$	$O(n \log(n))$	Coffman, et al., 1990
1 | $p_i = p$; s-batch | $\sum w_i C_i$	$O(n \log(n))$	Albers & Brucker, 1993
1 | $p_i = p$; s-batch; r_i | $\sum w_i C_i$	$O(n^{14})$	Baptiste, 2000
1 | s-batch | $\sum U_i$	$O(n^3)$	Brucker & Kovalyov, 1996
1 | $p_i = p$; s-batch | $\sum w_i U_i$	$O(n^4)$	Hochbaum & Landy, 1994
1 | $p_i = p$; s-batch; r_i | $\sum w_i U_i$	$O(n^{14})$	Baptiste, 2000
1 | $p_i = p$; s-batch; r_i | $\sum T_i$	$O(n^{14})$	Baptiste, 2000

Table 1.2: NP-hard s-Batch Problems (Brucker, 2004)

Problem	Reference
$1 \mid r_i$; s-batch $\mid L_{max}$	Lenstra & Rinnooy Kan, 1980
$1 \mid$ chains; s-batch $\mid \sum C_i$	Albers & Brucker, 1993
$1 \mid$ prec; s-batch $\mid \sum C_i$	Lawler, 1978
$1 \mid r_i$; s-batch $\mid \sum C_i$	Lenstra & Rinnooy Kan, 1980
$1 \mid$ s-batch $\mid \sum w_i C_i$	Albers & Brucker, 1993
$1 \mid$ chains; $p_i = 1$; s-batch $\mid \sum w_i C_i$	Albers & Brucker, 1993
$1 \mid$ chains; $p_i = 1$; s-batch $\mid \sum U_i$	Lenstra, 1977
$1 \mid$ s-batch $\mid \sum w_i U_i$	Karp, 1972
$1 \mid$ s-batch $\mid \sum T_i$	Du & Leung, 1990
$1 \mid$ chains; $p_i = 1$; s-batch $\mid \sum T_i$	Leung & Young, 1990

CHAPTER 2

THE s-BATCH PROBLEM UNDER AVERAGE WEIGHTED COMPLETION

Imagine a single photocopy machine. This copy machine is state-of-the-art, as it only needs the documents loaded onto the document scanning tray, and the machine does the rest. Placing a document onto the scanning tray is referred to as setup. Since the setup takes time, loading the machine requires a setup time. A single document may have a single page or many pages for copying. If a multi-page document is copied, the machine will fasten the loose pages together, otherwise any single page documents are left loose on the collating/output tray.

Now imagine a large set of documents, in an arbitrary order, to be copied. Some documents having just a single page, while others are multi-page documents. An inefficient method of completing this task would be to copy the documents one by one. This, of course, requires a setup time for each document to be copied. For productive purposes, the documents should be arranged in an order to utilize the machine, and time, efficiently. Also, an even more efficient method would be to batch certain documents together to be loaded at the same time. Since only one setup time is required for each batch, overall time is reduced. This photocopy machine example leads to the one machine batching problem. The one machine batching problem is defined as: "To find a sequence

of jobs and a collection of batches that partition this sequence of jobs such that the objective function is minimized." (Albers & Brucker, 1993)

Jobs

Using the photocopy machine example, each document to be copied can be viewed as a job. Each job has a time for completion; obviously it would take more time to copy a document containing ten pages than a document of one page. Also, each job has a priority, or weight. Perhaps some documents are being waited for, therefore they haves a higher priority. To generalize, there is a positive integer n number of jobs $J_i(i = 1, \ldots, n)$, with times $p_i(i = 1, \ldots, n)$, and weights $w_i(i = 1, \ldots, n)$, to be processed on one machine.

Batching

To improve the usage of the one machine, batching jobs together will increase job throughput efficiency (Albers & Brucker, 1993). A batch is a set of jobs which are processed jointly (Albers & Brucker, 1993). For example, it would be more efficient to copy a batch of single page documents, then a batch of multi-page documents requiring stapling, rather than have some multi-page documents mixed in with some single page documents. Batch size is the number of jobs contained in the batch. All jobs in a batch are not available until the last job in a batch is finished (Albers & Brucker, 1993). In other words, one could not remove any documents from the collating/output tray until the copy machine finished copying all documents set in the loading tray. Therefore, each job's completion time $C_i(i = 1, \ldots, n)$ is defined to be the completion time of the entire

batch $C_j (j = the\ j^{th}\ batch)$ (Albers & Brucker, 1993).

Weighted Average Completion Time

Recall our machine is state-of-the-art, and its only requirement is to load the document scanning tray. We can refer to this procedure as machine setup. Since time is required for setup, we can define this time as a constant where setup time $s \geq 0$. Set up time s is independent, so it does not depend on batch sequence, and it does not depend on the number of jobs in a batch (Albers & Brucker, 1993).

Using the batch completion times and priorities, the weighted average completion time for the schedule is defined as

$$F = \sum w_i C_i .$$

Here is an example of four documents needing to be copied. The first document has one page, the second has two pages, the third three pages, and finally the fourth a four page document. Also, for simplistic purposes, they all have the same priority. So, using a fixed job sequence $JS = J_1, J_2, J_3, J_4$, processing times $p_1 = 1, p_2 = 2, p_3 = 3, p_4 = 4$, and priorities of $w_1 = w_2 = w_3 = w_4 = 1$, Figure 2.1 shows a possible batched schedule with an objective function of $F = 29$.

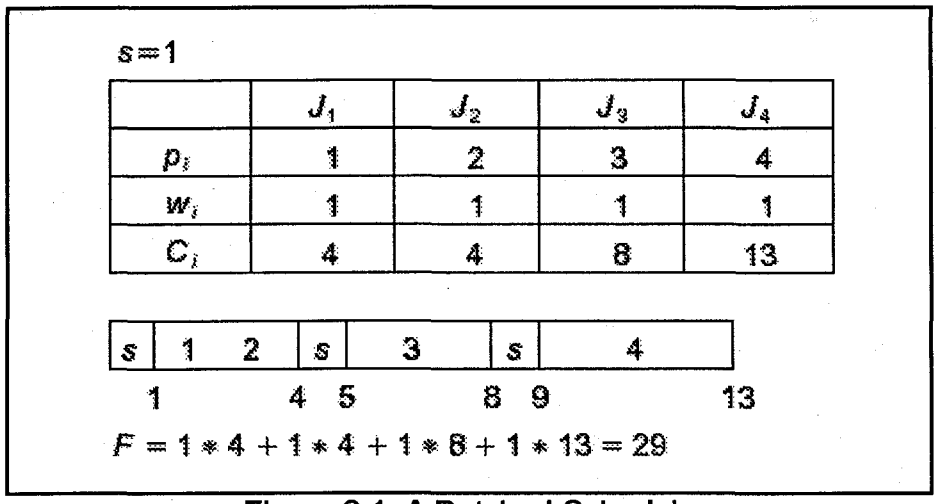

Figure 2.1: A Batched Schedule

NP-Completeness

The one machine batching problem is NP-hard (Albers & Brucker, 1993). This is shown by reducing the 3-Partitioning problem, which is already known to be NP-hard, to the one machine batching problem as proved by (Albers & Brucker, 1993).

3-Partition: Given a sequence of $3m$ positive integers $A = \{a_1, \ldots, a_{3m}\}$, and a positive integer B, such that

$$\frac{B}{4} < a_i < \frac{B}{2} \text{ and } \sum_{i=1}^{3m} a_i = m\,B,$$

for all $1 \le i \le 3m$. Can A be divided into m disjoint sequences $I_1, \ldots, I_m$ such that

$$\sum_{a_i \in I_j} a_i = B$$

for all $1 \le j \le m$ (Albers & Brucker, 1993)? Therefore, by reducing 3-Partition to the one machine s-batching problem, the s-batch problem under average weighted completion is NP-complete.

CHAPTER 3

THE 2-APPROXIMATION ALGORITHM

For any instance of an s-batch scheduling problem I, there is an algorithm that computes a solution that is no worse than twice the optimal schedule. As mentioned in Chapter 2, finding an optimal solution is NP-complete (Albers & Brucker, 1993). To apply an upper and lower bound for comparison against an approximate solution for I, a standardized structure is necessary to compute the bounds. This standardized structure is called the 'standard' order, in which the jobs of I are place in descending order according to priority. The priority of a job is the job's weight divided by the job's time. For example, if a job has a short processing time and a high weight, then that job has a high priority, and should be placed toward the front of the schedule.

Standard Order: Order jobs by priority. Given jobs $J_i(i = 1, \ldots, n)$, weights $w_i(i = 1, \ldots, n)$, and times $p_i(i = 1, \ldots, n)$ such that

$$\frac{w_i}{p_i} \geq \frac{w_{i+1}}{p_{i+1}}.$$

This process was done by placing the divided results into an array

$$A = \left\{\frac{w_i}{p_i}\right\}(i = 1, \ldots, n),$$

and then sorting A from greatest to least using merge sort.

Inversion: Given a standard order, an inversion is two values $\frac{w_i}{p_i}$ and $\frac{w_j}{p_j}$ with $i < j$ such that

$$\frac{w_i}{p_i} < \frac{w_j}{p_j}.$$

Lemma 3.1: Given a set of jobs $J_i (i = 1, \ldots, n)$, with times $p_i (i = 1, \ldots, n)$, priorities $w_i (i = 1, \ldots, n)$, and a setup time $s = 1$. Let

$$P_i = \sum_{j=1}^{i} p_j.$$

Then

$$\sum_{i=1}^{n} P_i w_i$$

is minimized if the jobs are ordered in standard order.

Proof: Assume an inversion of the standard order as such, $\cdots | J_i | J_j | \cdots$. Then

$$\frac{w_i}{p_i} < \frac{w_j}{p_j}.$$

Notice that

$$\frac{w_i}{p_i} < \frac{w_j}{p_j} \Rightarrow w_i p_j < w_j p_i \Rightarrow w_i p_j - w_j p_i < 0.$$

Now switch the inversion so that $\cdots | J_j | J_i | \cdots$ and

$$\frac{w_j}{p_j} < \frac{w_i}{p_i},$$

and the objective function differs by Δ where

$$\Delta = P_j w_j + (P_j + P_i) w_i - P_i w_i - (P_i + P_j) w_j$$

$$= P_j w_j + P_j w_i + P_i w_i - P_i w_i - P_i w_j - P_j w_j$$

$$= P_j w_i - P_i w_j .$$

Since $P_j w_i - P_i w_j < 0$, then

$$\sum_{i=1}^{n} P_i w_i$$

is minimized if the jobs are in standard order. □

Lemma 3.2: Let C_i be the completion times of the optimal schedule for the 1 | s-batch | $\sum w_i C_i$ problem. Also let the processing times $p_i (i = 1, \ldots, n)$ be given in standard order. Then the objective function is

$$\sum_{i=1}^{n} C_i w_i \geq \sum_{i=1}^{n} (P_i + 1) w_i .$$

Proof: Let

$$Q_i = \sum_{j \in B_i} p_j ,$$

where B_i is the set of all jobs such that job J_j is not after job J_i in an optimal schedule. For example, given a schedule of jobs in order of $|s|P_2|P_3|P_1|P_4|$, $Q_1 = p_2 + p_3 + p_1$, $Q_2 = p_2$, $Q_3 = p_2 + p_3$, and $Q_4 = p_2 + p_3 + p_1 + p_4$. Keep in mind that the optimal schedule has inversions. It is obvious that $C_i \geq Q_i + 1$ because of $s = 1$. Now apply the Lemma 3.1, and

$$\sum_{i=1}^{n} C_i w_i \geq \sum_{i=1}^{n} (Q_i + 1) w_i$$

$$= \sum_{i=1}^{n} Q_i w_i + \sum_{i=1}^{n} w_i \geq \sum_{i=1}^{n} P_i w_i + \sum_{i=1}^{n} w_i$$

$$= \sum_{i=1}^{n} (P_i + 1) w_i .$$

Therefore,

$$\sum_{i=1}^{n} C_i w_i \geq \sum_{i=1}^{n} (P_i + 1) w_i,$$

showing an optimal schedule cannot be better than the schedule of jobs in standard order, thus setting a lower bound. □

An Upper Bound

Pseudo Batch: Given a set of n jobs $J_i (i = 1, \ldots, n)$ with times $p_i (i = 1, \ldots, n)$, a setup time $s = 1$, variable t such that $t = \sum p_i$, and batch B the pseudo batch algorithm is as follows:

insert s into B;

insert J_1 into B;

$t = 0$;

for $i = 2, \ldots, n$

$t = t + p_i$;

if $t \geq 1$

insert s into B;

insert J_i into B;

$t = 0$;

else insert J_i into B;

The Approximation Algorithm: Order the jobs in standard order, and then make a schedule using Pseudo Batch. Call this Pseudo Priority (PP).

Let m be the number of batches up to job J_i in the schedule, let

$$P_i = \sum_{j=1}^{i} p_j,$$

and let $\hat{C}_i$ be the completion time of the pseudo batch. Given the setup time of $s = 1$, and the tally of the job times within a pseudo batch is at least 1, each pseudo batch has a completion time $\hat{C}_i \geq 2$, see Figure 3.1. Then $m \leq P_i + 1$.

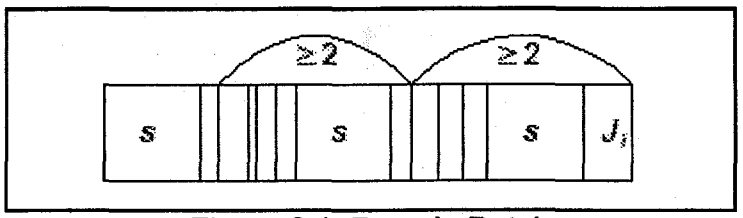

Figure 3.1: Pseudo Batch

Lemma 3.3: $\sum_{i=1}^{n} \hat{C}_i w_i \leq 2 \sum_{i=1}^{n} C_i w_i$.

Proof: Notice that $\hat{C}_i \leq P_i + m + 1$. Plug $m \leq P_i + 1$ into $\hat{C}_i \leq P_i + m + 1$, so $\hat{C}_i \leq P_i + P_i + 1 + 1 \leq 2P_i + 2$. Now the objective function of the schedule is,

$$\sum_{i=1}^{n} \hat{C}_i w_i \leq \sum_{i=1}^{n} (2P_i + 2) w_i \leq 2 \sum_{i=1}^{n} (P_i + 1) w_i \leq 2 \sum_{i=1}^{n} C_i w_i.$$

Therefore, ordering the jobs in PP, the upper bound will never be more than twice the the optimal schedule. □

An Alternative Bound

Using the Hire Fire Retire (HFR) algorithm, that is described in Chapter 4, an alternative bound can be computed using the following algorithm. Place the jobs

in standard order, and batch the jobs using the HFR algorithm. Call this algorithm Optimal Priority (OP). Note: the standard order is sorted using merge sort, so it has a time complexity of $O(n\log(n))$. Also, the HFR algorithm has a time complexity of $O(n\log(n))$, which is described in Chapter 4. Therefore, OP can be computed in $O(n\log(n))$.

Theorem 3.1: Algorithm PP is a 2 – Approximation.

Proof: Theorem 3.1 follows by Lemma 3.3.

Also we have:

Theorem 3.2: Algorithm OP is a 2 – Approximation.

Proof: Notice the cost of OP is better than the cost of PP.

The conjecture can be made that algorithm OP is much better that 2 – Approximation. However, a formal proof is not given of any bound in this thesis.

CHAPTER 4

EFFICIENT MONOTONE MATRIX SEARCHING

We first give intuition regarding monotone matrix searching. A totally monotone matrix has a special property that is useful for finding row minima in a time that is better than $O(n^2)$. Assume matrix $A[i,j]$ is totally monotone. Also assume row i minimum is found in column j. The row minimum for row $i+1$ is guaranteed not to be found in any column before j. For a monotone matrix, the row minima can be found in a diagonal pattern starting in the upper left corner, and tracing downward and to the right. Now more formally we define:

Monotone: Given a 2×2 matrix $A = \{a_{1,1}, a_{1,2}, a_{2,1}, a_{2,2}\}$, it is monotone if $a_{1,1} \geq a_{1,2}$ implies $a_{2,1} \geq a_{2,2}$ (Bein, Brucker, Larmore, & Park, 2004).

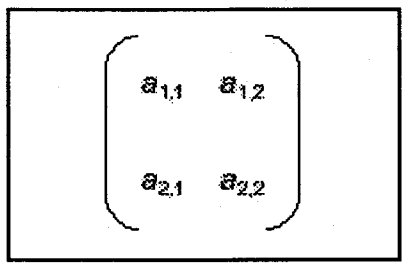

Figure 4.1: $a_{1,1} \geq a_{1,2} \rightarrow a_{2,1} \geq a_{2,2}$

Totally Monotone: A $m \times n$ matrix $A = \{a_{i,j}\}$, $i = 1, \ldots, m$ and $j = 1, \ldots, n$,

is totally monotone (TM) if every 2×2 sub-matrix of A is monotone (Bein, Brucker, Larmore, & Park, 2004). Using the monotonic property, TM implies that if $a_{i,j} > a_{i,k}$, then it cannot be that $a_{i+1,j} \leq a_{i+1,k}$ (Bein, Brucker, Larmore, & Park, 2004). Therefore, given row i minimum in column j, row $i+1$ minimum cannot be found in column $j-1$. For every minimum found at $a_{i,j}$, the remaining row and column portions of $[i \ldots n, j]$ and $[i, j \ldots n]$ are unnecessary, see Figure 4.2. Thus, finding row minima can be done in $O(n \log(n))$ time. However, this time complexity can be improved to $O(n)$ by using the SMAWK algorithm (Bein, Brucker, Larmore, & Park, 2004).

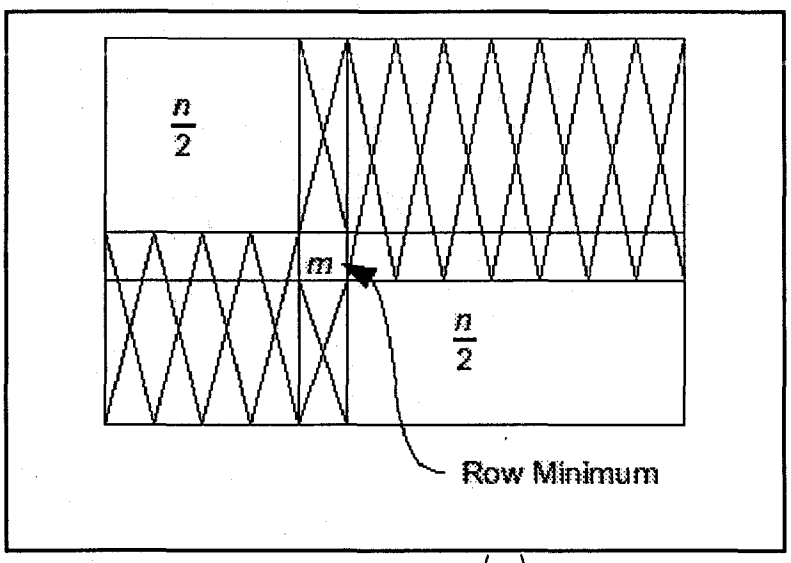

Figure 4.2: $T(n) = 2\,T\left(\frac{n}{2}\right) + \Theta(n)$

The Monge Property

Given an $m \times n$ matrix $A = \{c_{i,j}\}$, $i = 1, \ldots, m$ and $j = 1, \ldots, n$, A is Monge if $c_{i_1,j_1} + c_{i_2,j_2} \leq c_{i_2,j_1} + c_{i_1,j_2}$, see Figure 4.3 (Bein, Brucker, Larmore, &

Park, 2004). The Monge property implies the matrix is totally monotone (Bein, Brucker, Larmore, & Park, 2004).

Proof: Assume a 2×2 sub-matrix of A is not monotone, so $c_{1,1} \geq c_{1,2}$ and $c_{2,1} < c_{2,2}$. Then it would follow that $c_{1,1} + c_{2,2} > c_{2,1} + c_{1,2}$, which contradicts the Monge property. Therefore, the Monge property impiies A is totally monotone.

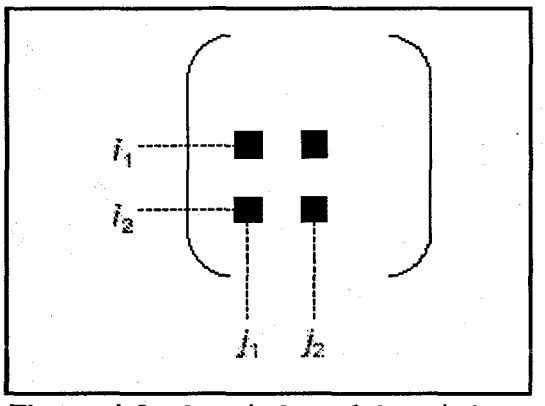

Figure 4.3: $c_{i_1,j_1} + c_{i_2,j_2} \leq c_{i_2,j_1} + c_{i_1,j_2}$

Table 4.1 shows some various examples of Monge matrices. The 'Matrix' coiumn gives the formuia that generates the matrix eiement $c[i,j]$. The 'Type' coiumn gives the type of Monge matrix the element creates. There are two different types of Monge matrices. The standard Monge type, labeled as M, implies the monotone matrix of $c[i,j] \geq c[i,j+1] \rightarrow c[i+1,j] \geq c[i+1,j+1]$, which will show the row minima from left to right, diagonally from top to bottom of the matrix. The reverse Monge type, labeled as RM, implies the monotone matrix of $c[i,j] \leq c[i,j+1] \rightarrow c[i+1,j] \leq c[i+1,j+1]$, which will show the row maxima from left to right, diagonally from top to bottom of the matrix.

Table 4.1: Monge Types

Matrix	Type
$c[i,j]=i+j$	M and RM
$c[i,j]=max\{i,j\}$	RM
$c[i,j]=min\{i,j\}$	M
$c[i,j]=max\{i,j\}-min\{i,j\}$	RM
$c[i,j]=i\cdot j$	RM
$c[i,j]=(i+j)^2$	RM
$c[i,j]=(i-j)^2$	M
$c[i,j]=h(i-j)$ for convex h	M
$c[i,j]=h(i-j)$ for concave h	RM
$c[i,j]=F(i,j)$ if all second derivatives are non-positive	M
$c[i,j]=F(i,j)$ if all second derivatives are non-negative	RM
$c[i,j]=\begin{cases}\int_{\alpha_i}^{\beta_j} f(y)dy & \alpha_i \le \beta_j \\ \int_{\beta_j}^{\alpha_i} g(y)dy & \beta_j < \alpha_i\end{cases}$	M
f and g are given non-negative integrabie functions with $f+g\ge 0$ and $\alpha_0 \le \ldots \alpha_n, \beta_0 \le \ldots \beta_n$ are real parameters.	

Negative to Positive Crossing Point

Given our matrix A is Monge, we can expioit the property to efficiently find a row w between any two columns $j_1 < j_2$ in which $c_{w,j_1} \le c_{w,j_2}$ and $c_{w+1,j_1} > c_{w+1,j_2}$. Row w is referred as the negative to positive crossing point, or in short, the crossing point because $c_{w,j_1} - c_{w,j_2} \le 0$ and $c_{w+1,j_1} - c_{w+1,j_2} > 0$. This crossing point is found using a binary searching technique. Figure 4.4 shows the crossing point between two columns. Let $A = c_{w,j_1}$, $B = c_{w,j_2}$, $C = c_{w+1,j_1}$, and $D = c_{w+1,j_2}$.

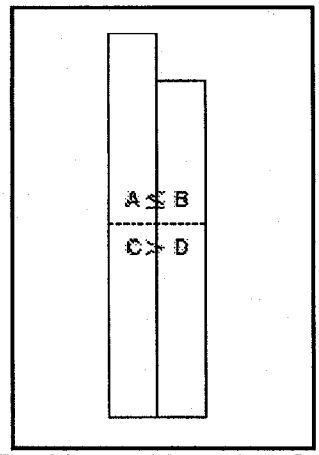

Figure 4.4: Positive to Negative Crossing Point

On-Line Matrix Indexing

Given a $n \times n$ 2-dimensional matrix $A[i, j]$, row indices are numbered such that $i = 1, \ldots, n$, and column indices are numbered such that $j = 0, \ldots, n-1$, see Figure 4.5. Numbering row indices starting at 1 simplifies the algorithm description and impiementation.

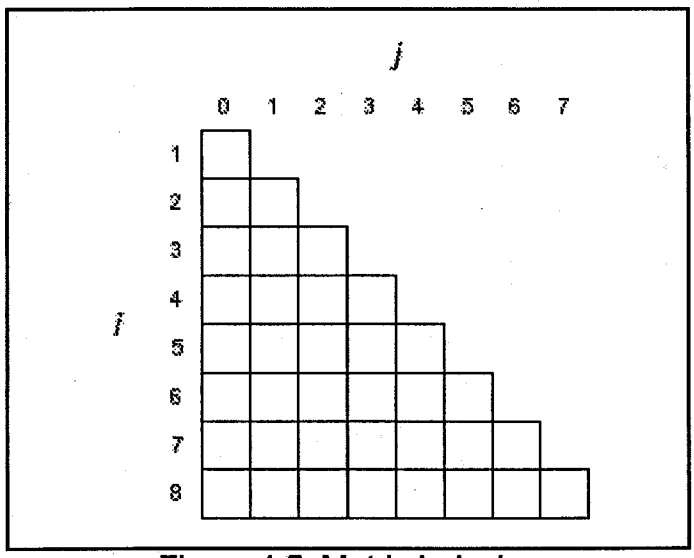

Figure 4.5: Matrix Indexing

On-Line Protocol

The on-line protocol, or on-line row column protocol (ORC) is used for the dynamic program that is explained in Chapter 5. ORC is described as, once row i minimum m_i has been found, column $j = i$ is available. Column availability is defined as, for every element $e_{i \ldots n,j}$ can be computed. Note for $j = 0$, a value of $m_0 = 0$ is used as a minimum for computing $e_{i \ldots n,0}$, see figure 4.6.

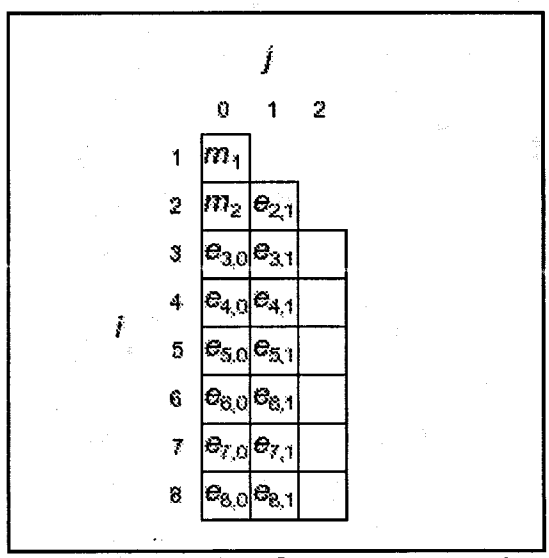

Figure 4.6: The On-Line Protocol

Hire Fire Retire Algorithm (HFR)

The purpose of this algorithm is to find the minimum of each matrix row in $O(n \log(n))$ time. First the algorithm will be given, then its details. To make the algorithm description easier to describe, some basic business company terms are used. These terms are: employee, staff, boss, potential boss, lackey, sackee, newbie, hire, fire, and retire. These terms will be defined as used in the scope of this algorithm, and then a description will be given of the algorithm's data

structures.

The HFR algorithm is as follows:

1. Hire $newbie = 0$;
2. Find row 1 minimum;
3. Hire $newbie = 1$;
4. Find row 2 minimum;
5. For $j = 2, \ldots, n - 1$ repeat

   5.1. Hire $newbie = j$;

   5.2. Fire as many sackees using newbie and lackeys as possible;

   5.3. Find row $i = j$ minimum;

   - If the boss does not contain the minimum, then the potential boss must contain the minimum, and retire the boss.

HFR Terminology

Employee: Using the dynamic program shown in Chapter 5, and the on-line protocol as a technique for iterating through the Monge matrix, entire columns are either used for finding row minima, or ignored. An employee is considered an entire column of the matrix. The column index j is used as the employee number.

Cost: A cost is a single element of the matrix.

Staff: A staff is the current collection of employee numbers. The staff must be kept in strict order from least to greatest.

Boss: An employee is the boss if and only if the employee is the first member of the staff.

Potential Boss: An employee is a potential boss if and only if the employee is the second member of the staff.

Newbie: An employee is a newbie if and only if the employee is the last member of the staff.

Sackee: An employee is a sackee if and only if the employee is the second from the last member of the staff.

Lackey: An employee is a lackey if and only if the employee is the third from the last member of the staff.

HFR Data Structure

Linked List: The staff is maintained by using a doubly linked list of employee numbers. The doubly linked list is necessary to keep a $O(1)$ time for staff operations. The list Staff operations are:

- isEmpty(): Returns true if the list is empty, otherwise returns false.
- print(): Prints a list of the current employee numbers to the screen.
- size(): Returns the current number of employee numbers in the list.
- hire(j): Appends employee number j to the end of the list.
- retire(): Removes the first employee number from the list.
- fire(): Removes the second to the last employee number from the list.
- getBoss(): Returns the first employee number of the list.
- getPotentialBoss(): Returns the second employee number of the list.
- getSackee(): Returns the second from the last employee number of the list.
- getLackey(): Returns the third from the last employee number of the

list.

HFR Operators

Hire: An employee is hired using the on-line matrix protocol. The new employee's number j is appended to the staff.

Fire: Firing an employee is done because it can be determined with certainty that the suspected employee could not possibly contain a row minimum. When an employee j is fired, the employee's number is removed from the staff.

Recall algorithm step 5.2.Fire as many sackees using newbie and lackeys as possible. The firing sequence is as follows:

1. Find the negative to positive crossing point at row x between lackey and sackee.
2. Find the negative to positive crossing point at row y between sackee and newbie.
3. If $x \geq y$, then fire sackee and try to fire another sackee with another lackey.

3.1.Else stop trying to fire a sackee.

Finding the negative to positive crossing points can be expanded from searching for this point between two employees lackey and sackee, then searching for the crossing point between sackee and newbie, to searching for the two points between the three employees simultaneously. This is done by a set of comparisons performed at row w between costs c such that $A = c_{w,lackey}$ and $B = c_{w,sackee}$, and between costs $B = c_{w,sackee}$ and $C = c_{w,newbie}$. Now the firing sequence is as follows:

1. If $A \leq B > C$, then fire sackee, and try again.

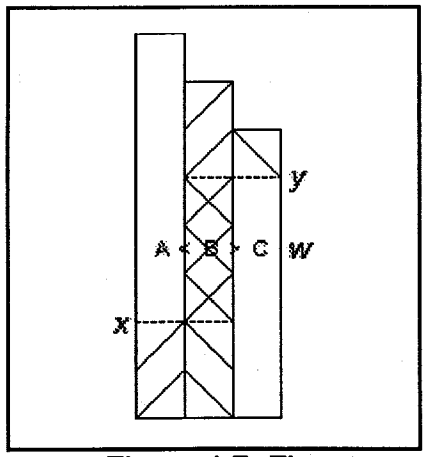

Figure 4.7: Fire

- The negative to positive crossing point at row x between lackey and sackee, and the negative to positive cross point at row y between sackee and newbie must be in positions such that $x \geq y$, see Figure 4.7.

2. If $A > B \leq C$, then do not fire sackee, and stop trying.

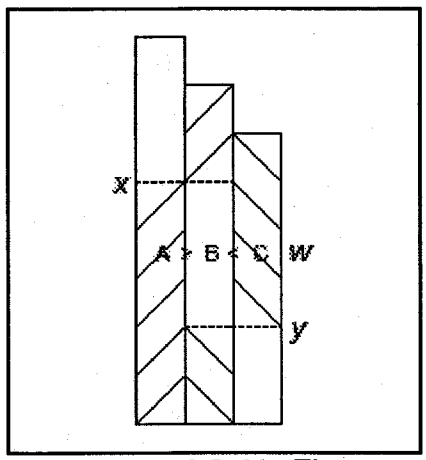

Figure 4.8: No Fire

- The negative to positive crossing point at row x between lackey and sackee, and the negative to positive crossing point at row y

between sackee and newbie must be in positions such that $x < y$, see Figure 4.8.

3. If $A \leq B \leq C$, then iook down.

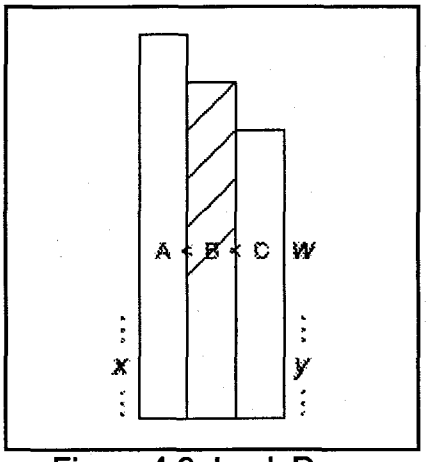

Figure 4.9: Look Down

➢ The negative to positive crossing point at row x between iackey and sackee, and the negative to positive crossing point at row y between sackee and newbie must be in positions such that $w < x$ and $w < y$, see Figure 4.9.

4. If $A > B > C$, then look up.

➢ The negative to positive crossing point at row x between lackey and sackee, and the negative to positive crossing point at row y between sackee and newbie must be in positions such that $w > x$ and $w > y$, see Figure 4.10.

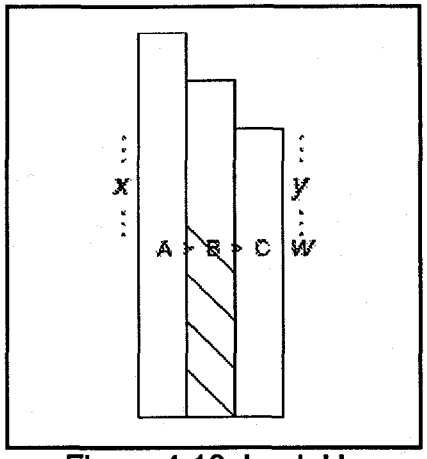

Figure 4.10: Look Up

In the event of equality between costs, a tie breaker occurs in the form of a coercive comparison, to give the costs a lexicographical order. See table 4.2.

Table 4.2: Lexicographical Coercive Comparisons

TEST	COERCIVE TEST	OPERATION
$A < B > C$	$A < B > C$	Fire
$A < B < C$	$A < B < C$	Look Down
$A < B = C$	$A < B < C$	Look Down
$A > B > C$	$A > B > C$	Look Up
$A > B < C$	$A > B < C$	No Fire
$A > B = C$	$A > B < C$	No Fire
$A = B > C$	$A < B > C$	Fire
$A = B < C$	$A < B < C$	Look Down
$A = B = C$	$A < B < C$	Look Down

Using Table 4.2, comparisons can be grouped according to their corresponding operations. Then the comparisons can be simplified into four basic comparisons used in the firing sequence. See table 4.3.

Table 4.3: Simplified Comparisons

	Fire	No Fire	Look Down	Look Up
	$A < B > C$	$A > B < C$	$A < B < C$	$A > B > C$
	$A = B > C$	$A > B = C$	$A < B = C$	
			$A = B < C$	
			$A = B = C$	
Simplified:	$A \leq B > C$	$A > B \leq C$	$A \leq B \leq C$	$A > B > C$

Find Row Minimum/Retire: An employee is retired if and only if the employee is the boss, and the boss no longer contains a row minimum. To find the row $i = j$ minimum, the sequence is as follows:

1. Set $min = c_{i,boss}$;
2. If $min < c_{i,potential\ boss}$, then return min;
   2.1. Else retire the boss, goto step 1.

HFR Testing

Figure 4.11 shows the HFR algorithm's output using a statically declared 18×18 matrix. Along with the hiring, firing, row minimum, and retiring reports, the output displays a graph of which matrix elements, or costs, are accessed, and the number of accesses per element.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	33																	
2	20	60																
3	24	62	56															
4	7	43	35	46														
5	13	47	37	46	36													
6	22	54	42	49	37	50												
7	13	43	29	34	20	31	47											
8	18	46	30	33	17	26	40	38										
9	31	57	39	40	22	29	41	37	48									
10	38	62	42	41	21	26	36	30	39	51								
11	43	65	43	40	18	21	29	21	28	38	44							
12	52	72	48	43	19	20	26	16	21	29	33	37						
13	77	95	69	62	36	35	39	27	30	36	38	40	45					
14	96	112	84	75	47	44	46	32	33	37	37	37	40	58				
15	114	128	98	87	57	52	52	36	35	37	35	33	34	50	62			
16	131	143	111	98	66	59	57	39	36	36	32	28	27	41	51	59		
17	164	174	140	125	91	82	78	58	53	51	45	39	36	48	56	62	63	
18	171	179	143	126	90	79	73	51	44	40	32	24	19	19	35	39	38	54

Off-Line Matrix

Hire	Fire	Retire	Row Min	Staff
0	-	-	33	(0)
1	-	-	20	(0,1)
2	1	-	24	(0,2)
3	2	-	7	(0,3)
4	3	-	13	(0,4)
5	-	-	22	(0,4,5)
6	-	-	13	(0,4,5,6)
7	6,5	0	17	(4,7)
8	-	-	22	(4,7,8)
9	-	-	21	(4,7,8,9)
10	9,8	-	18	(4,7,10)
11	10	4	16	(7,11)
12	-	-	27	(7,11,12)
13	-	-	32	(7,11,12,13)
14	13	7	33	(11,12,14)
15	14	11	27	(12,15)
16	15	-	36	(12,16)
17	16	-	19	(12,17)

HFR Output

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
1	1																		
2	1	1																	
3	1	3	2																
4	1	1	2	0															
5	1	1	1	2	1														
6	1	1	1	1	2	0													
7	2	0	1	2	3	0	0												
8	2	0	1	1	3	1	1	1											
9	0	1	2	1	1	0	1	2	0										
10	4	2	4	4	5	1	1	5	0	0									
11	4	1	2	3	7	1	0	6	0	0	0								
12	0	0	0	0	8	6	3	8	0	0	1	0							
13	0	0	0	0	5	6	3	8	3	1	2	2	0						
14	1	1	0	0	0	2	2	10	9	3	6	4	0	0					
15	1	1	0	0	0	2	2	10	11	4	4	7	4	0	0				
16	1	1	0	0	0	0	0	3	5	2	0	4	9	3	4	0			
17	2	2	0	0	0	0	0	0	0	0	0	0	11	6	7	4	2		
18	2	2	0	0	0	0	0	0	0	0	0	0	13	6	8	5	5	1	
	25	18	16	14	35	19	13	53	28	10	13	17	37	15	19	9	7	1	349

Number of Off-Line Matrix Element Accesses

Figure 4.11: Hire Fire Retire Algorithm Run

CHAPTER 5

AN $O(n \log(n))$ ALGORITHM FOR THE BATCHING PROBLEM WITH FIXED ORDER

Shortest Path Problem Reduction: Given a fixed order of jobs, the one machine batching problem can be solved in polynomial time by reducing it to a shortest path problem. Using such a fixed order of jobs $J_i (i = 1, \ldots, n)$, with processing times $p_i (i = 1, \ldots, n)$, priorities $w_i (i = 1, \ldots, n)$, and a setup time of $s = 1$, a schedule is defined as

$$JS = J_{i_1} | s\, J_{i_1+1} \cdots J_{i_2} | s\, J_{i_2+1} \cdots J_{i_3} | \cdots | s\, J_{i_k+1} \cdots J_{i_{k+1}}.$$

The processing time of the j^{th} batch is defined as

$$P_j = s + \sum_{\mu = i_j + 1}^{i_{j+1}} p_\mu .$$

The following is an example of a schedule with three batches:

$$J_{i_1} | s\, J_{i_1+1} \cdots J_{i_2} | s\, J_{i_2+1} \cdots J_{i_3} | s\, J_{i_3+1} \cdots J_{i_4}.$$

Then

$$P_1 = 1 + \sum_{\mu = i_1 + 1}^{i_2} p_\mu ,\ P_2 = 1 + \sum_{\mu = i_2 + 1}^{i_3} p_\mu , \text{ and } P_3 = 1 + \sum_{\mu = i_3 + 1}^{i_4} p_\mu .$$

It is important to point out that the weighted cost of each batch is dependent on the previous batches, except for the first batch. Keeping with the example, the weighted cost of the first batch is

$$\left(\sum_{\mu=i_1+1}^{i_2} w_\mu\right) P_1 .$$

The weighted cost of the second batch is

$$\left(\sum_{\mu=i_2+1}^{i_3} w_\mu\right) P_1 + \left(\sum_{\mu=i_2+1}^{i_3} w_\mu\right) P_2 .$$

Also the weighted cost of the third batch is

$$\left(\sum_{\mu=i_3+1}^{i_4} w_\mu\right) P_1 + \left(\sum_{\mu=i_3+1}^{i_4} w_\mu\right) P_2 + \left(\sum_{\mu=i_3+1}^{i_4} w_\mu\right) P_3 .$$

Finally, an objective function can be computed for the example as

$$\left(\sum_{\mu=i_1+1}^{i_4} w_\mu\right) P_1 + \left(\sum_{\mu=i_2+1}^{i_4} w_\mu\right) P_2 + \left(\sum_{\mu=i_3+1}^{i_4} w_\mu\right) P_3 .$$

Therefore, an objective function for JS can be written as

$$\sum_{i=1}^{n} C_i w_i = \sum_{j=1}^{k} \left(\sum_{\mu=i_j+1}^{i_{j+1}} w_\mu\right) P_j .$$

Now the schedule can be represented as a path in a directed acyclic graph (DAG). Figure 5.1 shows an example of such a graph for four jobs. Each edge of the graph $c[i_j, i_{j+1}]$ has a cost defined as the cost of the j^{th} batch. Using the example from above,

$$c[i_1, i_2] = \left(\sum_{\mu=i_1+1}^{n} w_\mu\right) P_1 , \; c[i_2, i_3] = \left(\sum_{\mu=i_2+1}^{n} w_\mu\right) P_2 , \text{ and } c[i_3, i_4] = \left(\sum_{\mu=i_3+1}^{n} w_\mu\right) P_3 .$$

Notice $c[i_1, i_2] + c[i_2, i_3] + c[i_3, i_4]$ is the same as the objective function for this example. The processing time of the j^{th} batch is

$$P_j = \left(s + \sum_{\mu=i_j+1}^{i_{j+1}} p_\mu\right).$$

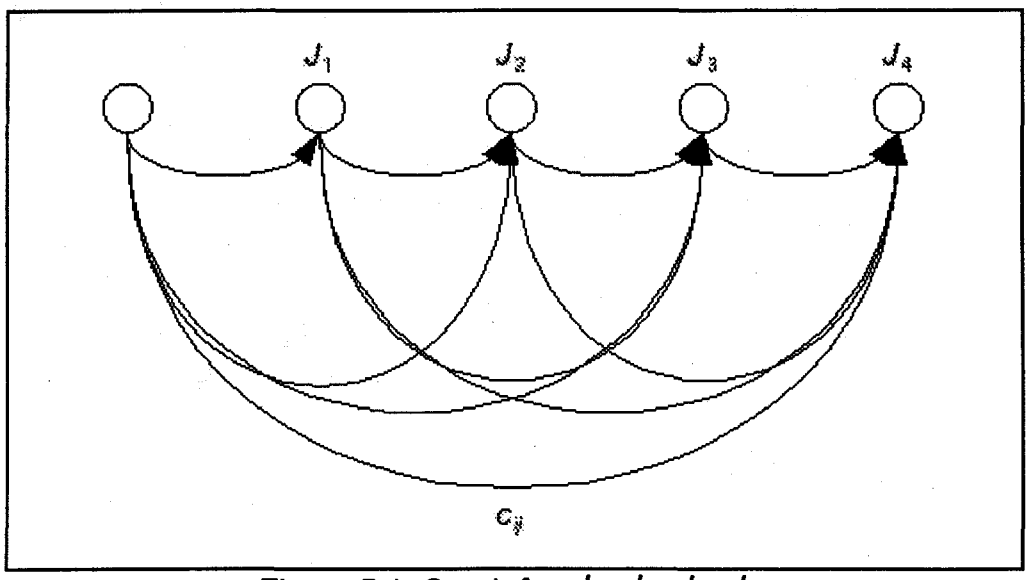

Figure 5.1: Graph for J_1, J_2, J_3, J_4

To generalize, the cost of an edge from i to j is calculated as

$$c_{ij} = \left(\sum_{\mu=i+1}^{n} w_\mu \right) \left(s + \sum_{\mu=i+1}^{j} p_\mu \right) = (W_n - W_i)(s + P_j - P_i).$$

Lemma 5.1: The matrix representation of the DAG is Monge.

Proof: For a positive integer n, given are $p_0, w_0, \ldots, p_n, w_n$ with $p_0 = 0$, $w_0 = 0$, and $p_1, w_1 \geq 0, \ldots, p_n, w_n \geq 0$, as well as $s > 0$. Assume $s = 1$. Let

$$P_k = \sum_{i=0}^{k} p_i \text{ and } W_k = \sum_{i=0}^{k} w_i \text{ for } k = 1, \ldots, n.$$

Consider the matrix

$$C = \{c[i,j]\}_{i=0,\ldots,1,\ j=1,\ldots,n} \text{ with } c[i,j] = (W_n - W_i)(s + P_j - P_i),$$

so

$$C = \begin{bmatrix} c_{i,j}=(W_n-W_i)(s+P_j-P_i) & c_{i,j+1}=(W_n-W_i)(s+P_{j+1}-P_i) \\ c_{i+1,j}=(W_n-W_{i+1})(s+P_j-P_{i+1}) & c_{i+1,j+1}=(W_n-W_{i+1})(s+P_{j+1}-P_{i+1}) \end{bmatrix}.$$

To show C is Monge, it is necessary to show that

$$c_{i,j} + c_{i+1,j+1} \leq c_{i,j+1} + c_{i+1,j}, \text{ or } (c_{i,j} - c_{i,j+1}) + (c_{i+1,j+1} - c_{i+1,j}) \leq 0.$$

$$(c_{i,j} - c_{i,j+1}) + (c_{i+1,j+1} - c_{i+1,j})$$

$$= (W_n - W_i)(P_j - P_{j+1}) + (W_n - W_{i+1})(P_{j+1} - P_j)$$

$$= -P_j(W_n - W_i) + P_j(W_n - W_{i+1})$$

$$= P_j(W_n - W_{i+1} - W_n + W_i) = -P_j(W_{i+1} - W_i) = -P_j W_i \leq 0$$

Therefore C is Monge. □

Dynamic Program

Now that the batching problem with fixed order has been shown to reduce to a shortest path problem, a dynamic program is used to find the shortest path from the first DAG node to the last. Recall the matrix indexing from Chapter 4, so the dynamic program matrix uses indices of $i = 1, \ldots, n$ for rows and $j = 0, \ldots, n-1$ for columns. The matrix is defined as $e(i,j) = E[j] + c_{j,i}$ where $E[j] = min_{0 \leq l < j}\{e[j,l]\}$.

		j			
	0	1	2	$E[0]=0$
1	$E[0]+c_{0,1}$			$E[1]=min_{0\leq k<1}\{E[k]+c_{k,1}\}$
i 2	$E[0]+c_{0,2}$	$E[1]+c_{1,2}$		$E[2]=min_{0\leq k<2}\{E[k]+c_{k,2}\}$
3	$E[0]+c_{0,3}$	$E[1]+c_{1,3}$	$E[2]+c_{2,3}$	$E[3]=min_{0\leq k<3}\{E[k]+c_{k,3}\}$

Figure 5.2: A Dynamic Program for a 3x3 Matrix

Figure 5.2 shows an example of a matrix $E = e(i,j)$ for $n = 3$. Following the example, E is formalized to

$$e(0,0) = 0$$
$$e(i,j) = min_{0 \leq l < j}\{e(j,l) + c[j,i]\}$$

Notice in Figure 5.2 that when $E[j]$ is known, column $j = i$ is available. It should be clear that the dynamic program can use the ORC protocol as described in Chapter 4. It should also be pointed out that E is Monge if c is Monge. This can be shown in a similar manner that was used in Lemma 5.1. However, rather than giving a formal proof, the remark is made that the Monge property is preserved under the operations of minimum and addition.

Dynamic Program Testing

For testing purposes the following results was used: An optimal value can be compared with the following: If all n jobs have processing time $p_1, \ldots, p_n = 1$ with setup time $s = 1$, and if

$$n = \frac{m(m+1)}{2},$$

then we can describe the value of an optimal batching schedule with

$$m(m+1)\frac{3m^2 + 11m + 10}{24}$$

(Bein, Epstein, Larmore, & Noga, 2004).

Then a set of

$$n = \{120, 465, 1830, 7260, 28920, 64980\}$$

jobs were used with processing times $p_1, \ldots, p_n = 1$, weights $w_1, \ldots, w_n = 1$, and a setup time $s = 1$. The dynamic program reports an optimal value by returning the minimum of the last row of the matrix.

$m=15$ $n=120$ $m(m+1)\frac{3m^2+11m+10}{24}=8500$ Program output: Size of N: 120 Number of MATRIX lookups: 2342 N log N = 828.827 Factor: 2.82568 Optimal Score: 8500	$m=30$ $n=465$ $m(m+1)\frac{3m^2+11m+10}{24}=117800$ Program output: Size of N: 465 Number of MATRIX lookups: 11932 N log N = 4120.41 Factor: 2.89583 Optimal Score: 117800
$m=60$ $n=1830$ $m(m+1)\frac{3m^2+11m+10}{24}=1749175$ Program output: Size of N: 1830 Number of MATRIX lookups: 57363 N log N = 19832.9 Factor: 2.89232 Optimal Score: 1749175	$m=120$ $n=7260$ $m(m+1)\frac{3m^2+11m+10}{24}=26940650$ Program output: Size of N: 7260 Number of MATRIX lookups: 269216 N log N = 93115 Factor: 2.89122 Optimal Score: 26940650
$m=240$ $n=28920$ $m(m+1)\frac{3m^2+11m+10}{24}=422834500$ Program output: Size of N: 28920 Number of MATRIX lookups: 1229429 N log N = 428588 Factor: 2.86856 Optimal Score: 422834500	$m=360$ $n=64980$ $m(m+1)\frac{3m^2+11m+10}{24}=2126849550$ Program output: Size of N: 64980 Number of MATRIX lookups: 2981879 N log N = 1.03888e+06 Factor: 2.87028 Optimal Score: 2126849550

Figure 5.3: Dynamic Program Test Results

CHAPTER 6

A GENETIC ALGORITHM

A genetic algorithm (GA) gives a searching technique used to find approximate solutions for optimization problems. Such optimization problems include scheduling and routing. A GA has three basic operators: selection, crossover, and mutation. Later in the chapter, a 5 step GA will be described along with the operator descriptions. But first, in order to better understand the algorithm, a list of GA terms are defined.

Genetic Algorithm Terminology

Chromosome: A chromosome is one possible solution for a given problem (Mitchell, 1996). Usually the solution is represented as a string of bits. However, it is also convenient to use a simple array of indexes that represent a mapping to a solution candidate.

Gene: A gene is a component of the chromosome that represents a single element of the solution candidate (Mitchell, 1996).

Locus: Locus is an exact location of a gene in a given chromosome (Mitchell, 1996).

Allele: An allele is the value of a gene at a given locus in a particular chromosome (Mitchell, 1996).

Genome/Genotype: The genome, or genotype, is the data structure used to represent a chromosome, with each element of the genome representing the locus of a gene (Mitchell, 1996).

Population: A population is a set of chromosomes. For the GA, the population size is predetermined, and passed as a parameter to the algorithm. Each chromosome of the population is randomly chosen (Mitchell, 1996).

Parent: A parent is a chromosome selected for mating with another chromosome, or the other parent. The selection of a parent from the population is based on the fitness score given to each chromosome by the fitness function (Mitchell, 1996).

Offspring: The offspring are the children of the chosen parents. Exactly two offspring will be generated from a given set of parents, and the offspring are based on the crossover (Mitchell, 1996).

Fitness Function: Also known as an objective function, the fitness function is a key element of the GA. It is the bases of whether a chromosome will continue as a member of the population in future generations, or be deleted. The fitness function is also used as a deciding factor for selection in crossover mating (Mitchell, 1996). Since each chromosome of a population must be 'scored' by the objective function, it is important that the function be computed efficiently with a good time complexity.

Genetic Algorithm Operators

Selection: Selection is the process of choosing chromosomes for crossover mating. This selection is based on the fitness of the chromosome, therefore, the

fitter the chromosome, the better chance it has of being selected for reproduction (Mitchell, 1996). An example of a selection method is the roulette wheel analogy. Using a chromosomes fitness score to associate a likely hood of being selected, think of the pockets of the roulette wheel varying in size, based on this probability. If the ball falls into a chromosome's pocket, that chromosome is selected. To chose N chromosomes is to play N games of roulette.

Crossover: Crossover is the process of generating two offspring using a pair of chromosomes as parents (Mitchell, 1996). A locus is randomly chosen for the parents, then the sequences are exchanged before and after the locus for the

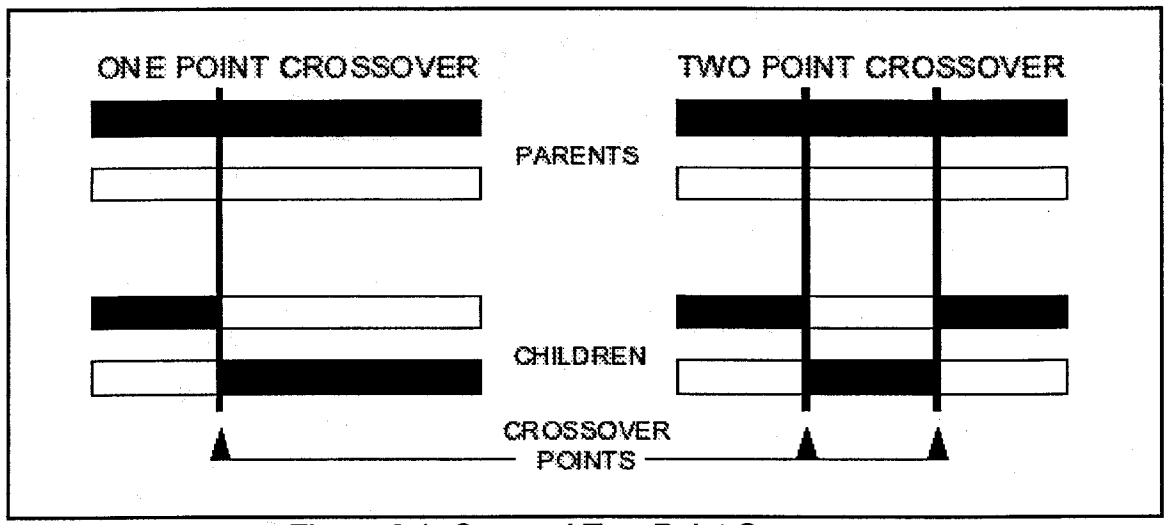

Figure 6.1: One and Two Point Crossover

new children. Figure 6.1 shows examples of crossover for a single selected locus and a dual selected locus. Crossover will occur based on a probability, or crossover rate, which is predetermined and passed to the GA as a parameter. The crossover rate is usually set to be large. A typical crossover rate would be $p_c = 0.7$ (Mitchell, 1996).

Mutation: Mutation is a probable change at each locus of a given chromosome (Mitchell, 1996). A chromosome gene will mutate only if the probability of mutation, or mutation rate, allows it. The mutation rate is predetermined, passed to the GA as a parameter, and is usually set to be very small; typically $p_m = 0.001$ (Mitchell, 1996). The purpose of mutation is to avoid the GA from falling into a local minima, which could occur if all of the chromosomes where to become too similar.

A 5 Step Genetic Algorithm

A problem's solution candidate can easily be represented with a string of bits. Then a GA would operate as follows; stated from (Mitchell, 1996):

1. Start with a randomly generated population of n l-bit chromosomes (candidate solutions to a problem).
2. Calculate the fitness $f(x)$ of each chromosome x in the population.
3. Repeat the following steps until n offspring have been created:

   a) Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.

   b) With probability p_c (the "crossover probability" or "crossover rate"), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents. (Note that here the crossover rate is defined to be the probability that two parents

will cross over in a single point. There are also "multi-point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place.)

c) Mutate the two offspring at each locus with probability p_m (the mutation probability or mutation rate), and place the resulting chromosomes in the new population.

If n is odd, one new population member can be discarded at random.

4. Replace the current population with the new population.
5. Go to step 2.

A generation is one iteration of the GA. A run is one complete cycle through a given number of generations. Given the randomness of the GA, two runs of the same problem may yield two different solutions (Mitchell, 1996). It is generally a good idea to take an average of the results from several runs of the same problem.

Genetic Algorithm Library for C++

As previously mentioned, finding an optimal schedule for the s-batch problem under weighted average completion time is NP-complete. The next best solution to optimal is near-optimal, or an approximation. A GA was used to find an approximation in the search space. The GA for this project is a C++ library called GAlib, that was developed by Mathew Wall of the Massachusetts Institution of Technology (Wall, 2005). The library offers practical and extensible classes for applications using a GA where optimization is critical. GAlib includes some simple default GA models with default genome types and operators for quick

applications. However, if one wishes to implement a complex solution, GAlib is very customizable. GAlib is capable of being installed on several platforms such as UNIX, MacOS, Windows 9x/NT/2K/XP, and DOS/Windows 3.1 (Wall, 2005). For this project, GAlib was built on the UNIX platform using the g++ compiler.

Even though GAlib supplies many GA models, a simple GA, which is GAlib's default model and was described previously in this chapter, was used for implementation of this project. When GAlib uses the simple model, the population is completely replaced by crossover and mutation of the previous generation's population, after each generation (Wall, 2005).

Two primary classes are involved when using the library, a genome class and a genetic algorithm class. An instance of a genome class represents a single possible solution of the problem. Then the genetic algorithm class specifies how the genomes will evolve. Using these classes, there are three basic entities one must define to implement a solution using a GA (Wall, 2005).

1. Define a representation.
2. Define the genetic operators.
3. Define the objective function.

GAlib provides many functions to aid in implementing 1 and 2, with little to no modifications, which greatly simplifies the usage of GAlib. However, 3 must be programed by the user. The HFR algorithm is used to define the objective function for the implementation of the simple GA used for this project.

Genetic Algorithm Object

The GA object determines the evolution of the generations of populations.

GAlib performs the process of evolution by determining, through selection, which pairs of genomes to mate, which genomes to replace, and which genomes will survive during the current generation. When an instance of the GA object is created, a population is initialized. Then until termination, for each generation, genomes are selected for mating. For each pair of selected genomes, crossover occurs to generate two new offspring. The offspring are mutated if the mutation rate decides mutation should take place. Finally the offspring is then placed into the current population, replacing any genomes with worse scores. Usually, the user of GAlib decides the requirement for termination, and usually that requirement is a predetermined number of generations.

Population Object

All of the genomes that make a population are contained in a population object (Wall, 2005). This object also calculates and stores statistics about the population. Some of these statistics are deviation, best genome, and average fitness. An important method of the population object is to decide which genomes are selected for mating (Wall, 2005).

Defining A Representation

Since the s-batch problem just utilizes job indices, in which times and weights are assigned to each job, using a simple one dimensional array for schedule representation is sufficient. GAlib has a template class, that is derived from the GA genome class, called GA1DArrayGenome (Wall, 2005). This class is a dynamic array of objects. A predefined population size of these genomes is

declared, along with a predefined genome length, of which is size n. During initialization of each genome (or chromosome), the job indices are used as alleles, which are randomly chosen, and assigned to each locus of the chromosome.

Defining The Genetic Operators

The genetic operators does the work of evolution for each generation (Wall, 2005). GAlib utilizes three primary operators, initialization, mutation, and crossover. For this project, an initializer was implemented by the user so that each genome is randomly initialized in linear time. To do this, a source array $A = \{1, ..., n\}$ of job indices is generated. Then, an index i of A is randomly selected, and the value of i is assigned to locus j of the genome for $j = 0, ..., n-1$. As indices of A are selected, A shrinks by placing the value of A_n into the empty slot. The default mutation of GAlib is used, and is performed based on the mutation probability. Lastly, a predefined crossover is used, simply because certain elements of this project make the list order based crossover of GAlib convenient to use. When the order based crossover mates two genomes together, it does not allow any gene duplication.

Defining The Objective Function

A GA only needs a single score to determine how good a chromosome when compared to the other chromosomes of the population (Wall, 2005). Selection, for mutation and mating purposes, is based on a fitness score. The fitness score is based on the objective score that is returned, as a floating point value, from the

objective function.

"It is important to note the distinction between fitness and objective scores. The objective score is the value returned by your objective function; it is the raw performance evaluation of a genome. The fitness score, on the other hand, is a possibly-transformed rating used by the genetic algorithm to determine the fitness of individuals for mating. The fitness score is typically obtained by a linear scaling of the raw objective scores." (Wall, 2005)

As previously mentioned, the user is to define the objective function for GAlib. The HFR algorithm is used to determine an objective score for each genome of the current population for each generation. The actual value returned by the objective function for this project, is the reported minimum for row n.

CHAPTER 7

GAs-BATCH EXPERIMENT

Four tests were performed to find a near optimal schedule given a set of n jobs with respective times and priorities. Job processing times, $p_i (i = 1, \ldots, n)$, were randomly chosen as real numbers between 0 and 2, excluding 0 to avoid an illegal divide during boundary calculations. Job weights, $w_i (i = 1, \ldots, n)$, were randomly chosen integers between 1 and 3. The experiment used a genetic algorithm library GAlib, written by Mathew Wall of MIT. The genetic algorithm begins by initializing a population by randomly selecting a fixed schedule. Each fixed schedule is then given a score using an objective function, which is the HFR algorithm described in Chapter 4. Then appropriate crossover mating and mutation occurs within the population for each generation. The best genome score is reported at certain intervals of generations; either every 100^{th} generation or every 1000^{th} generation. Each test was allowed three runs, with slightly different parameters. All tests had the following parameters in common:

- Number of jobs $n = 100$.
- Crossover rate $p_c = 0.85$.
- Mutation rate $p_m = 0.005$.

The tests population size, and number of generations are as follows:

1. Generations: 1000, Population: 100

2. Generations: 1000, Population: 500
3. Generations: 5000, Population: 500
4. Generations: 5000, Population: 1000

The following figures contain the results for the best run of an experiment. For every run, an upper bound, a lower bound, and the OP were calculated, and reported, for the given set of jobs. A percentage is given of how far from the lower bound the score of the best permutation is, as reported by the GA.

Job #	Time	Weight	Job #	Time	Weight
1	0.5206	2	51	0.1981	1
2	1.6986	3	52	0.4955	3
3	0.5605	3	53	1.2885	3
4	0.9909	1	54	0.9622	1
5	0.6627	1	55	1.0279	3
6	1.6407	3	56	0.2198	1
7	1.7567	2	57	0.1857	1
8	0.7364	1	58	0.7334	3
9	0.3230	2	59	0.0884	1
10	1.9437	3	60	0.5797	2
11	0.4776	1	61	1.1130	1
12	1.7822	3	62	1.2216	1
13	1.8396	1	63	1.5254	1
14	1.6600	1	64	1.4773	3
15	1.2882	2	65	1.5481	2
16	0.1151	2	66	0.0689	1
17	0.7665	3	67	0.0370	3
18	1.8352	3	68	0.6845	2
19	0.4980	1	69	0.3084	3
20	1.3071	2	70	1.2801	2
21	0.0070	2	71	0.2395	1
22	1.4429	1	72	1.6273	2
23	1.2838	3	73	0.0618	2
24	0.0354	2	74	1.9481	1
25	1.5993	2	75	0.5557	3
26	1.9621	1	76	0.6094	1
27	1.1795	2	77	1.1720	2
28	1.1843	3	78	1.2712	3
29	1.7196	3	79	0.9732	3
30	1.4565	1	80	1.7556	3
31	0.3918	3	81	1.9838	1
32	0.8165	2	82	0.2675	2
33	1.3852	2	83	0.3888	2
34	0.4852	3	84	1.3463	1
35	1.4778	2	85	0.5986	1
36	1.0439	1	86	0.0335	3
37	0.7401	3	87	0.1262	2
38	1.2485	3	88	1.2632	1
39	0.6660	2	89	0.0215	3
40	1.6307	3	90	0.3383	3
41	1.2630	2	91	1.7391	2
42	0.3422	2	92	0.2458	3
43	1.3372	1	93	0.7786	3
44	1.4616	2	94	1.6084	3
45	0.3707	2	95	0.2656	1
46	0.0646	2	96	0.2730	3
47	0.4896	2	97	1.1884	1
48	1.9207	2	98	0.2701	2
49	1.5328	3	99	1.4924	1
50	1.7579	3	100	0.2743	2

Raw Data

Generation #	Best Score
100	9,140.38
200	8,951.24
300	8,765.39
400	8,883.60
500	8,658.44
600	8,526.46
700	8,411.15
800	8,365.98
900	8,365.98
1,000	8,208.30

GA Run

|S|67 83 90 46 69 34 3 55 71 21 87 100 56 59 31 76
|S|52 86 89 24 16 1 27 9 51 50 92
|S|75 60
|S|93 45 39 57
|S|73 58
|S|79 23 15 82 2 35 12
|S|42 96 25 41 10
|S|47 98
|S|95 28 94 80
|S|78 8 5 32 53 65 77 63
|S|88 85
|S|7 40 13 18 97 38 37
|S|48 14 68 36 81 22 64
|S|62 19 70 30 54 4
|S|66 33 11
|S|20 26
|S|43 17
|S|44 91 6
|S|99
|S|72 49 29 74
|S|84
|S|61|

s-Batch Schedule

Upper Bound: 12635.6
Optimal Priority Score: 12171.8
GA Best Score: 8208.3
Lower Bound: 5682.48
36.3266% away from Lower Bound.

s-Batch Schedule Fitness

Figure 7.1: Experiment 1

Job #	Time	Weight	Job #	Time	Weight
1	0.6980	3	51	0.8908	3
2	0.5071	2	52	0.1183	3
3	0.6779	3	53	1.6488	3
4	0.3681	2	54	1.2726	1
5	1.8432	1	55	1.0431	2
6	0.5274	2	56	1.3951	3
7	1.6636	3	57	1.9027	2
8	0.7772	1	58	1.0873	2
9	1.4492	1	59	0.1567	3
10	0.7715	3	60	0.0476	3
11	1.7246	1	61	1.6370	1
12	1.5973	3	62	1.9125	3
13	0.0141	2	63	1.1631	1
14	1.2447	2	64	1.4929	3
15	1.3968	3	65	0.2943	1
16	1.8605	2	66	1.1296	2
17	0.6310	2	67	0.5993	3
18	0.4577	3	68	1.4112	1
19	1.7845	2	69	0.4068	3
20	0.7560	2	70	0.9062	1
21	1.4653	2	71	0.1632	2
22	0.6264	2	72	1.9921	1
23	1.7112	3	73	1.8858	3
24	0.8435	1	74	0.2054	3
25	1.7277	3	75	1.7719	1
26	0.2162	3	76	0.3991	3
27	1.0461	2	77	1.2769	2
28	0.3829	3	78	1.1902	2
29	1.6714	1	79	1.7670	2
30	1.3393	2	80	1.3631	2
31	1.3561	1	81	1.2395	1
32	1.1630	2	82	1.2463	2
33	1.5674	2	83	1.7405	2
34	1.2960	1	84	0.0868	1
35	0.4465	1	85	1.3412	3
36	0.9795	3	86	1.5972	3
37	1.3353	3	87	1.2943	2
38	1.0407	3	88	1.5704	2
39	1.0093	1	89	0.6687	2
40	0.2070	1	90	1.2227	2
41	1.4991	1	91	0.0288	3
42	0.1893	3	92	1.8495	3
43	0.2367	3	93	1.9333	2
44	1.6053	3	94	1.7457	3
45	0.1998	2	95	1.7227	3
46	0.9763	3	96	1.7615	2
47	0.1442	3	97	1.6014	2
48	1.5429	1	98	0.8009	2
49	1.7600	1	99	1.6653	1
50	1.0635	1	100	1.4891	1

Raw Data

Generation #	Best Score
100	10,924.70
200	10,570.40
300	10,475.50
400	10,335.10
500	10,120.50
600	9,966.96
700	9,875.67
800	9,834.13
900	9,797.30
1,000	9,731.60

GA Run

|S|28 2 18 40 10 74 47 26 13 43 76 52 3 65 71 27
|S|51 64 45
|S|60
|S|67 4 6 46 12 53 90 92
|S|59
|S|69 55 36 35 89 91 15 42 23 1
|S|20 22 38 88
|S|77
|S|39 100 84 7 98 58 25 17 95 83
|S|37
|S|14 33
|S|96 93 32 16 82 87
|S|57 21
|S|99 31 78 50 11 9
|S|44 86 62
|S|81 85 79 29
|S|41 56 24 94
|S|30
|S|49
|S|34 5 73 8
|S|66 63
|S|54 80
|S|70 72 61
|S|68
|S|48
|S|97 19
|S|75|

s-Batch Schedule

Upper Bound: 14684.1
Optimal Priority Score: 13181.9
GA Best Score: 9731.6
Lower Bound: 7163.79
34.1448% away from Lower Bour

s-Batch Schedule Fitness

Figure 7.2: Experiment 2

Job #	Time	Weight	Job #	Time	Weight
1	1.4723	2	51	1.3558	1
2	1.8870	2	52	1.7750	2
3	1.2853	2	53	1.3052	2
4	0.7184	1	54	0.8435	2
5	1.0012	2	55	0.3513	3
6	1.7627	2	56	1.4792	1
7	0.3403	2	57	0.5633	2
8	0.5925	2	58	1.4387	1
9	0.7079	1	59	1.8687	2
10	1.4038	1	60	0.8654	3
11	1.3698	3	61	0.7533	1
12	0.5951	3	62	1.4328	3
13	1.2328	3	63	0.2217	2
14	0.0050	2	64	1.3325	1
15	1.7997	3	65	1.1605	1
16	1.2894	2	66	1.1871	1
17	0.1257	1	67	1.4463	3
18	0.0553	3	68	0.3575	2
19	0.1126	2	69	0.6695	3
20	1.7960	3	70	1.6516	2
21	1.0479	1	71	0.2718	3
22	0.1276	1	72	0.1314	1
23	0.5169	1	73	1.5429	3
24	0.7116	2	74	0.7515	3
25	0.9842	3	75	0.1386	3
26	1.2654	1	76	1.2655	1
27	1.5688	1	77	0.8267	3
28	0.5235	1	78	0.1745	2
29	1.5958	3	79	1.4296	3
30	1.0551	1	80	1.0046	2
31	1.5939	1	81	0.5229	1
32	0.6674	3	82	1.1064	1
33	1.5207	1	83	1.1538	2
34	1.1306	2	84	0.7206	2
35	1.4963	1	85	0.6650	3
36	0.7471	3	86	0.8441	1
37	0.4854	1	87	1.5029	1
38	1.9324	1	88	1.7718	3
39	1.5949	1	89	0.5457	3
40	0.2272	2	90	0.4575	3
41	0.7593	2	91	0.1401	3
42	0.9330	1	92	1.5763	1
43	1.3532	1	93	0.4212	1
44	1.1298	1	94	0.0152	3
45	1.9851	1	95	1.2578	3
46	0.3444	2	96	0.4408	1
47	0.5292	3	97	1.2114	3
48	1.2986	3	98	1.7498	1
49	1.7345	3	99	1.0111	2
50	1.6833	3	100	0.1884	3

Raw Data

Generation #	Best Score
100	9,138.16
500	8,711.31
1,000	8,627.66
1,500	8,627.66
2,000	8,627.66
2,500	8,627.66
3,000	8,627.66
3,500	8,627.66
4,000	8,627.66
4,500	8,627.66
5,000	8,627.66

GA Run

|S|72 47 89 74 91 7 77 40 94 100 75 68 36 25 24 12 16 80 95 55
|S|93 22 66 15 18 78 63 46 85 14 48 82 28 65
|S|88 60 11 71
|S|87 50 19
|S|27 97 73
|S|30 79 17 83 20 2 41
|S|32 42 59 5 62 49
|S|4 54
|S|90 33
|S|81 39 86
|S|29 64 3 8
|S|56 58
|S|69 26 44 52 34 84 9
|S|61 57
|S|21 99 1
|S|67 96
|S|92 38 53 35 10 23
|S|43
|S|70 45 31 13
|S|37 51
|S|98 6
|S|76|

s-Batch Schedule

Upper Bound: 12662.9
Optimal Priority Score: 11925.5
GA Best Score: 8627.66
Lower Bound: 5776.06
41.4064% away from Lower Bou

s-Batch Schedule Fitness

Figure 7.3: Experiment 3

Job #	Time	Weight	Job #	Time	Weight
1	0.2722	1	51	0.5848	3
2	1.6789	3	52	1.5856	1
3	1.7256	2	53	0.0804	1
4	1.2755	3	54	1.1824	1
5	1.4722	2	55	0.6506	3
6	1.9200	2	56	1.8823	1
7	1.2205	1	57	1.1425	2
8	1.3335	1	58	0.4053	3
9	1.9777	2	59	0.7607	1
10	0.5422	3	60	1.2667	2
11	0.5540	3	61	1.9024	1
12	0.4561	2	62	0.9458	2
13	0.6582	3	63	0.1634	2
14	1.7128	3	64	0.3440	2
15	0.1353	3	65	0.3950	3
16	0.6228	2	66	1.2024	3
17	0.1844	2	67	0.4024	1
18	0.6841	2	68	1.6157	1
19	0.9181	1	69	0.3737	3
20	1.1683	3	70	1.1461	2
21	1.0516	3	71	1.0001	3
22	0.9114	2	72	0.3468	1
23	0.4954	1	73	1.3197	2
24	1.0649	3	74	0.6023	3
25	0.6361	2	75	1.3092	1
26	1.9208	3	76	1.0835	3
27	0.4108	1	77	1.5677	1
28	0.0603	2	78	0.2758	1
29	1.4898	1	79	0.8024	3
30	1.1935	2	80	1.0757	1
31	1.7002	3	81	1.2796	3
32	0.0326	2	82	0.1771	3
33	0.1181	2	83	1.0661	1
34	1.7946	2	84	0.1648	2
35	0.6764	3	85	0.5908	1
36	1.4265	1	86	0.4641	2
37	0.0511	3	87	0.3392	1
38	0.0000	1	88	1.7336	2
39	0.5829	2	89	1.5629	1
40	1.0196	2	90	1.7392	1
41	0.6719	3	91	0.2048	2
42	0.4091	1	92	0.0837	3
43	1.3172	3	93	0.8521	3
44	0.1518	1	94	1.5035	3
45	0.2424	3	95	0.9791	1
46	0.4321	3	96	0.8518	3
47	1.1617	2	97	0.8574	2
48	1.8273	3	98	1.9408	1
49	1.1813	1	99	0.8216	3
50	0.3160	3	100	1.3633	1

Raw Data

Generation #	Best Score
100	8,577.57
500	8,180.34
1,000	7,600.96
1,500	7,593.00
2,000	7,593.00
2,500	7,593.00
3,000	7,593.00
3,500	7,593.00
4,000	7,593.00
4,500	7,593.00
5,000	7,593.00

GA Run

|S|51 12 15 74 28 10 92 50 17 44 18 13 69 53 58 45 11 84 46
|S|82 38 33 65 55 41
|S|91 32 99 4 67 57 37 27
|S|72
|S|71 43 52 60 48
|S|93 87 42 40 66 22
|S|86 25
|S|79 14 63 76
|S|24 62
|S|96 88 2 16 36 49 47
|S|81
|S|97 73 64
|S|75 30 31 9 21 94 80 100
|S|23 59 78 20 26 83 19 34 85
|S|54 39 77 29
|S|70
|S|90 8
|S|56 35 6 5 3 1
|S|95 68 98
|S|61 7
|S|89 |

s-Batch Schedule

Upper Bound: 12473.5
Optimal Priority Score: 10777.2
GA Best Score: 7593
Lower Bound: 5357.96
31.4105% away from Lower Boun

s-Batch Schedule Fitness

Figure 7.4: Experiment 4

CHAPTER 8

CONCLUSION

In this thesis paper, the s-batch problem was described along with descriptions of heuristics and approximations for the s-batch problem. Finding an optimal schedule for a set of jobs, with processing times and priorities, is NP-complete (Albers & Brucker, 1993). However, given a fixed order of jobs, a batch schedule can be computed in $O(n \log(n))$ time. This was accomplished by reducing the one machine s-batch problem to a shortest path problem. It was also shown that the matrix representation of the DAG is a Monge matrix. To exploit this property, a dynamic program was created to find the shortest path in a way such that the HFR algorithm, along with the on-line protocol, could be used.

A 2-approximation algorithm was given and its solution used for an upper bound. A lower bound was proved. An alternative approximation algorithm, OP, was given which used the $O(n \log(n))$ matrix searching technique. This algorithm is better than PP, but we conjecture that it has an approximation ratio which is close to 1 if n is very large. This thesis neither formalizes this nor gives any proof of such conjecture. However, future work will focus on such results.

For experimentation, a set of jobs were generated with random values as times and weights. Then a population of randomly selected permutations was

generated. A genetic algorithm was used to find the best permutation out of the set of originally generated permutations. The HFR algorithm was used as the objective function for the genetic algorithm.

Using a set of jobs of size $n = 100$ the genetic algorithm created solutions with an average of 35.8% away from the lower bound. It was also found that the genetic algorithm did not need a large set of permutations or generations to converge to this average.

Instead of using approximations of the type described here, future work could also employ linear programming techniques. The main issue here is to define appropriate integer linear programming formulations.

APPENDIX

SOURCE CODE

```
/*****************************************************************************************/
/*        GASbatch.C
/*****************************************************************************************/
using namespace std;

#include "personnel.h"

#include <iostream.h>
#include <math.h>

#include <ga/GASimpleGA.h>
#include <ga/GA1DArrayGenome.h>

struct Job {
  int n;
  float p;
  int w;
  float q;
  float P;
};

const int NJ = 5;
const int N = NJ + 1;
const int S = 1;
Job jobs[NJ];
Job temp[NJ];
float P[N];
int W[N];

/** GA support functions **/
void myInitializer(GAGenome &);
float myObjective(GAGenome &);

/** support functions **/
void initJobs();
float fRand();
float myRand(float, float);
int beinRand(int, int);
void pSum(const float*);
```

```
void wSum(const int*);
void fire(empID, Personnel&, float*);
float getRowMin(empID, Personnel&, float*);
float getRowElement(int, int, float);
float Cij(int, int);
void initQ();
void sortQ();
void mergeSort(Job [], Job [], int, int);
void merge(Job [], Job [], int, int, int);
float getLowerBound();
float getUpperBound();
void printMySchedule(GAGenome &);

int main(int *argc, char *argv[]) {
  const int Generations = atoi(argv[1]);
  const int Population = atoi(argv[2]);
  const float P_crossover = 0.85;
  const float P_mutation = 0.005;
  float best_score;
  float upper_bound;
  float lower_bound;
  float op_genome_score;
  float away;

  srand(time(NULL));

  initJobs();

  GA1DArrayGenome<int> genome(NJ, myObjective);
  genome.initializer(myInitializer);
  genome.crossover(GA1DArrayGenome<int>::OrderCrossover);

  GASimpleGA ga(genome);
  ga.populationSize(Population);
  ga.nGenerations(Generations);
  ga.pMutation(P_mutation);
  ga.pCrossover(P_crossover);
  ga.minimize();
  ga.initialize();

  for(int i = 1; i < Generations; i++) {
    if (i % 100 == 0) {
      cout << "Generation #" << i << ' ';
      cout << " best score: " << ga.statistics().bestIndividual().score();
      cout << endl;
    }
    ga.step();
  }
```

```
  cout << "Generation #" << Generations;
  cout << " best score: " << ga.statistics().bestIndividual().score();
  cout << endl;

  for (int i = 0; i < NJ; i++) {
    cout << jobs[i].n << ": " << jobs[i].p << ", " << jobs[i].w << endl;
  }

  initQ();
  sortQ();

  GA1DArrayGenome<int> best_genome(NJ);
  GA1DArrayGenome<int> op_genome(NJ);

  // initialize the op_genome
  for (int i = 0; i < NJ; i++) {
    op_genome.gene(i, jobs[i].n);
  }

  best_genome = ga.statistics().bestIndividual();

  best_score = best_genome.score();

  upper_bound = getUpperBound();
  lower_bound = getLowerBound();
  op_genome_score = myObjective(op_genome);
  away = (best_score - lower_bound) / (upper_bound - lower_bound);

  cout << "Upper Bound:          " << upper_bound << endl;
  cout << "Optimal Priority Score: " << op_genome_score << endl;
  cout << "GA Best Score:        " << best_score << endl;
  cout << "Lower Bound:          " << lower_bound << endl;
  cout << away * 100 << "% away from Lower Bound." << endl;

  printMySchedule(best_genome);

  return(0);
}

void myInitializer(GAGenome & g) {
  int index;
  int source[NJ];

  for (int i = 0; i < NJ; i++) {
    source[i] = i + 1;
  }

  GA1DArrayGenome<int>& genome = (GA1DArrayGenome<int>&)g;
```

```
  for (int i = NJ - 1; i >= 0; i--) {
    index = beinRand(0, i);
    genome.gene(i, source[index]);
    source[index] = source[i];
  }
}

float myObjective(GAGenome & g) {
  Personnel staff;
  empID newbie;
  float current_p[N];
  int current_w[N];
  float E[N];
  int index;

  GA1DArrayGenome<int>& genome = (GA1DArrayGenome<int>&)g;

  current_p[0] = 0.0;
  current_w[0] = 0;
  int j = 1;
  for (int i = 0; i < genome.length(); i++) {
    index = genome.gene(i);
    current_p[j] = jobs[index-1].p;
    current_w[j] = jobs[index-1].w;
    j++;
  }

  pSum(current_p);
  wSum(current_w);

  for (int i = 0; i < N; i++) {
    cout << P[i] << ' ';
  }

  for (int i = 0; i < N; i++) {
    cout << W[i] << ' ';
  }

  E[0] = 0.0;
  newbie = 0;

  // hire first column
  staff.hire(newbie);
  E[newbie+1] = getRowMin(newbie, staff, E);

  // hire second column
  newbie++;
  staff.hire(newbie);
```

```
  E[newbie+1] = getRowMin(newbie, staff, E);

  // run the algorithm for the rest of the columns
  for (newbie = 2; newbie < N-1; newbie++) {
    // hire a newbie
    staff.hire(newbie);

    // newbie tries to fire employees with imediate seniority using a lackey
    fire(newbie, staff, E);

    // find row min associated with newbie
    E[newbie+1] = getRowMin(newbie, staff, E);
  }

  return E[NJ];
}

void initJobs() {
  for (int i = 0; i < NJ; i++) {
    jobs[i].n = i + 1;
    jobs[i].p = myRand(0.00000000001, 2.0);
    jobs[i].w = beinRand(1, 3);
  }
}

float fRand() {
  return rand() / (float(RAND_MAX) + 1);
}

float myRand(float min, float max) {
  return fRand() * (max - min) + min;
}

int beinRand(int lower, int upper) {
  return (lower + rand() % (upper - lower + 1));
}

void pSum(const float* p) {
  P[0] = 0.0;
  // p[0] = 0 because of dummy job
  for (int i = 1; i < N; i++) {
    P[i] = p[i] + P[i-1];
  }
}

void wSum(const int* w) {
  W[0] = 0;
  // w[0] = 0 because of dummy job
```

```
  for (int i = 1; i < N; i++) {
    W[i] = w[i] + W[i-1];
  }
}

void fire(empID newbie, Personnel& staff, float* Ek) {
  bool firing;
  bool trying;
  empID sackee; // employee with imediate seniority to newbie
  empID lackey; // employee with imediate seniority to sackee
  float newbie_cost;
  float sackee_cost;
  float lackey_cost;
  int low;
  int high;
  int mid;

  firing = true;
  while (firing && staff.size() > 2) {
    sackee = staff.getSackee();
    lackey = staff.getLackey();
    low = newbie;
    high = N - 1;
    trying = true;

    while (trying) {
      mid = (low + high) / 2;
      newbie_cost = getRowElement(newbie, mid+1, Ek[newbie]);
      sackee_cost = getRowElement(sackee, mid+1, Ek[sackee]);
      lackey_cost = getRowElement(lackey, mid+1, Ek[lackey]);

      if ((lackey_cost <= sackee_cost) && (sackee_cost > newbie_cost)) {
          staff.flre();
          trying = false;
      }
      else if ((lackey_cost > sackee_cost) && (sackee_cost <= newbie_cost)) {
          firing = false;
          trying = false;
      }
      else if ((lackey_cost <= sackee_cost) && (sackee_cost <= newbie_cost)) {
          if (low >= high) {
            staff.flre();
            trying = false;
          }
          else {
            low = mid + 1;
          }
      }
```

```
      else if ((lackey_cost > sackee_cost) && (sackee_cost > newbie_cost)) {
        if (low >= high) {
          staff.fire();
          trying = false;
        }
        else {
          high = mid - 1;
        }
      }
    }
  }
}

float getRowMin(emplD newbie, Personnel& staff, float* Ek) {
  emplD boss = staff.getBoss();
  float min = getRowElement(boss, newbie+1, Ek[boss]);

  if (staff.size() == 1) {
    return min;
  }

  emplD poten_boss = staff.getPotentialBoss();

  if (min < getRowElement(poten_boss, newbie+1, Ek[poten_boss])) {
    return min;
  }

  staff.retire();
  return getRowMin(newbie, staff, Ek);
}

float getRowElement(int i, int j, float Ek) {
  // Cij + E[i]
  return (Cij(i, j) + Ek);
}

float Cij(int i, int j) {
  // Cij = (Wn - Wi)(S + (Pj - Pi))
  return ((W[N-1] - W[i]) * (S + (P[j] - P[i])));
}

void initQ() {
  for (int i = 0; i < NJ; i++) {
    jobs[i].q = (jobs[i].w / jobs[i].p);
  }
}

void sortQ() {
```

```
  mergeSort(jobs, temp, 0, NJ - 1);
}

void mergeSort(Job jobs[], Job temp[], int left, int right) {
  int mid;

  if (right > left) {
    mid = (right + left) / 2;
    mergeSort(jobs, temp, left, mid);
    mergeSort(jobs, temp, mid + 1, right);
    merge(jobs, temp, left, mid + 1, right);
  }
}

void merge(Job numbers[], Job temp[], int left, int mid, int right) {
  int left_end;
  int num_elements;
  int tmp_pos;

  left_end = mid - 1;
  tmp_pos = left;
  num_elements = right - left + 1;

  while ((left <= left_end) && (mid <= right)) {
    if (jobs[left].q >= jobs[mid].q) {
      temp[tmp_pos] = jobs[left];
      tmp_pos += 1;
      left += 1;
    }
    else {
      temp[tmp_pos] = numbers[mid];
      tmp_pos += 1;
      mid += 1;
    }
  }

  while (left <= left_end) {
    temp[tmp_pos] = numbers[left];
    left += 1;
    tmp_pos += 1;
  }

  while (mid <= right) {
    temp[tmp_pos] = numbers[mid];
    mid += 1;
    tmp_pos += 1;
  }
```

```
  for (int i = 0; i < num_elements; i++) {
    numbers[right] = temp[right];
    right -= 1;
  }
}

float getLowerBound() {
  float lb = 0.0;

  // compute completion times c for each job
  jobs[0].P = jobs[0].p;
  for (int i = 1; i < NJ; i++) {
    jobs[i].P = jobs[i-1].P + jobs[i].p;
  }

  // compute the lower bound
  for (int i = 0; i < NJ; i++) {
    lb = lb + ((jobs[i].P + 1) * jobs[i].w);
  }

  return lb;
}

float getUpperBound() {
  int j;
  int schedule_length;
  float ub = 0.0;
  float tally = 0.0;
  float C_hat[NJ];
  float schedule[2*NJ]; // worst case, one job per batch; m=n

  // make the pseudo batch schedule
  schedule[0] = -1;
  schedule[1] = jobs[0].p;
  j = 2;
  for (int i = 1; i < NJ; i++) {
    tally = tally + jobs[i].p;
    if (tally >= 1.0) {
      schedule[j] = -1;
      j++;
      schedule[j] = jobs[i].p;
      j++;
    }
    else {
      schedule[j] = jobs[i].p;
      j++;
    }
  }
```

```
  schedule_length = j;

  j = 0;
  float batch_time = 0.0;
  for (int i = 0; i < schedule_length;) {
    if (schedule[i] == -1) {
      i++;
      batch_time = S + batch_time;
      int batch_size = 0;
      while (schedule[i] != -1 && i < schedule_length) {
          batch_time = batch_time + schedule[i];
          i++;
          batch_size++;
      }
      for (int k = 0; k < batch_size; k++) {
          C_hat[j] = batch_time;
          j++;
      }
    }
  }

  for (int i = 0; i < NJ; i++) {
    ub = ub + (C_hat[i] * jobs[i].w);
  }

  return ub;
}

void printMySchedule(GAGenome & g) {
  Personnel staff;
  empID newbie;
  int index;
  float current_p[N];
  int current_w[N];
  float E[N];
  int batches[N];

  GA1DArrayGenome<int>& genome = (GA1DArrayGenome<int>&)g;

  current_p[0] = 0.0;
  current_w[0] = 0;
  int j = 1;
  for (int i = 0; i < genome.length(); i++) {
    index = genome.gene(i);
    current_p[j] = jobs[index].p;
    current_w[j] = jobs[index].w;
    j++;
  }
```

```
  pSum(current_p);
  wSum(current_w);

  E[0] = 0.0;
  newbie = 0;

  // hire first column
  staff.hire(newbie);
  E[newbie+1] = getRowMin(newbie, staff, E);
  batches[newbie] = staff.getBoss();

  // hire second column
  newbie++;
  staff.hire(newbie);
  E[newbie+1] = getRowMin(newbie, staff, E);
  batches[newbie] = staff.getBoss();

  // run the algorithm for the rest of the columns
  for (newbie = 2; newbie < N-1; newbie++) {
    // hire a newbie
    staff.hire(newbie);

    // newbie tries to fire employees with imediate seniority using a lackey
    fire(newbie, staff, E);

    // find row min associated with newbie
    E[newbie+1] = getRowMin(newbie, staff, E);
    batches[newbie] = staff.getBoss();
  }
  batches[NJ] = -1;

  int start_batch = 0;
  for (int finish_batch = 0; finish_batch < NJ; finish_batch++) {
    while (batches[finish_batch] == batches[finish_batch+1]) {
      finish_batch++;
    }
    cout << endl << "|S|";
    for (int j = start_batch; j <= finish_batch; j++) {
      cout << genome.gene(j) << ' ';
    }
    cout << "\b";
    start_batch = finish_batch + 1;
  }
  cout << '|' << endl << endl;
}
```

```
/*****************************************************************************************/
/*        personnel.h                                                                    */
/*****************************************************************************************/
typedef int empID;

struct NodeType;

class Personnel {
 public:
  bool isEmpty() const;
  void print() const;
  int size() const;
  void hire(empID assign_id);
  void retire();
  void fire();
  empID getBoss();
  empID getPotentialBoss();
  empID getSackee();
  empID getLackey();
  Personnel();                          // constructor
  Personnel(const Personnel& otherList); // copy constructor
  ~Personnel();                         // deconstructor
 private:
  NodeType* head;
  NodeType* tail;
  int length;
};

/*****************************************************************************************/
/*        personnel.C                                                                    */
/*****************************************************************************************/
#include "personnel.h"
#include <iostream.h>

typedef NodeType* NodePtr;
struct NodeType {
  empID id;
  NodePtr f_link;
  NodePtr b_link;
};

bool Personnel::isEmpty() const {
  return (head == NULL && tail == NULL);
}

void Personnel::print() const {
  NodePtr currPtr = head;
```

```
  cout << '(';
  while (currPtr != NULL) {
    cout << currPtr -> id;
    currPtr = currPtr -> f_link;
    if (currPtr != NULL) {
      cout << ',';
    }
  }
  cout << ')' << endl;
}

int Personnel::size() const {
  return length;
}

void Personnel::hire(empID assign_id) {
  NodePtr newNodePtr;

  newNodePtr = new NodeType;
  newNodePtr -> id = assign_id;

  if (length == 0) {
    newNodePtr -> b_link = NULL;
    newNodePtr -> f_link = NULL;
    head = newNodePtr;
    tail = newNodePtr;
  }
  else {
    tail -> f_link = newNodePtr;
    newNodePtr -> b_link = tail;
    newNodePtr -> f_link = NULL;
    tail = newNodePtr;
  }

  length++;
}

void Personnel::retire() {
  if (isEmpty()) {
    cout << "Error: personnel list is empty." << endl;
    exit(0);
  }

  NodePtr tempPtr = head;

  if (length == 1) {
    head = NULL;
    tail = NULL;
```

```
  }
  eise {
    head = tempPtr -> f_link;
    head -> b_link = NULL;
  }
  delete tempPtr;
  length--;
}

void Personnel::fire() {
  if (length < 3) {
    cout << "Error: cannot fire, staff size less than 3." << endl;
    exit(0);
  }
  NodePtr newbie = tail;
  NodePtr sackee = newbie -> b_link;
  NodePtr lackey = sackee -> b_link;

  newbie -> b_link = sackee -> b_link;
  lackey -> f_link = sackee -> f_link;
  delete sackee;
  length--;
}

empID Personnel::getBoss() {
  if (isEmpty()) {
    cout << "Error: personnel list is empty." << endl;
    exit(0);
  }
  return head -> id;
}

empID Personnel::getPotentialBoss() {
  if (isEmpty()) {
    cout << "Error: personnel list is empty." << endl;
    exit(0);
  }
  else if (length < 2) {
    cout << "No potential boss exists." << endl;
    exit(0);
  }

  return head -> f_link -> id;
}

empID Personnel::getSackee() {
  if (isEmpty()) {
    cout << "Error: personnel list is empty." << endl;
```

```
    exit(0);
  }
  else if (length < 3) {
    cout << "No sackee exists." << endl;
    exit(0);
  }

  return tail -> b_link -> id;
}

empID Personnel::getLackey() {
  if (isEmpty()) {
    cout << "Error: personnel list is empty." << endl;
    exit(0);
  }
  else if (length < 3) {
    cout << "No lackey exists." << endl;
    exit(0);
  }

  return tail -> b_link -> b_link -> id;
}

Personnel::Personnel() {
  head = NULL;
  tail = NULL;
  length = 0;
}

Personnel::~Personnel() {
  while (!isEmpty()) {
    retire();
  }
}
```

BIBLIOGRAPHY

Albers, S., & Brucker, P. (1993). The complexity of one-machine batching problems. *Discrete Applied Mathematics, Combinatorial Algorithms, Optimization and Computer Science, 47(2),* 87 – 107.

Baptiste, P. (2000). Batching identical jobs. *Mathematical Methods of Operations Research, 53(3),* 355 – 367.

Bein, W.W., Brucker, P., Larmore, L.L., & Park, J.K. (2004). The algebraic Monge property and path problems. *Discrete Applied Mathematics, 145(2005),* 455 – 464.

Bein, W.W., Epstein, L., Larmore L.L., & Noga, J. (2004). Optimally competitive list batching. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory,* (SWAT 2004), July 2004. LNCS 3111 Springer-Verlag, 77 – 89.

Brucker, P. (2004). *Scheduling Algorithms* (4th ed.). Berlin, Heidelberg, New York: Springer – Verlag.

Brucker, P., & Kovalyov, M.Y. (1996). Single machine batch scheduling to minimize the weighted number of late jobs. *Mathematical Methods of Operations Research, 43(1),* 1 – 8.

Coffman, E.G., Jr., Yannakakis, M., Magazine, M.J., & Santos, C. (1990). Batch sizing and job sequencing on a single machine. *Annals of Operations Research, 26(1 – 4),* 135 – 147.

Du, J., & Leung, J.Y.-T. (1990). Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research, 15(3),* 483 – 495.

Hochbaum, D.S., & Landy, D. (1994). Scheduling with batching: minimizing the weighted number of tardy jobs. *Operations Research Letters, 16(2),* 79 – 86.

Karp, R.M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J Watson Res. Center, Yorktown Heights, N.Y., 1972),* (pp. 85 – 103). Plenum, New York.

Lawler, E.L. (1978). Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics, 2,* 75 – 90.

Lenstra, J.K. (1977). *Sequencing by enumerative methods.* Mathematical Centre Tracts. 69.

Lenstra, J.K., & Rinnooy Kan, A.H.G. (1980). Complexity results for scheduling chains on a single machine. *European Journal of Operational Research, 4(4),* 270 – 275.

Leung, J.Y.-T., & Young, G.H. (1990). Preemptive scheduling to minimize mean weighted flow time. *Information Processing Letters, 34(1),* 47 – 50.

Mitchell, M. (1996). *An introduction to genetic algorithms.* Massachusetts: MIT Press.

Ng, C.T., Cheng, T.C.E., & Yuan, J.J. (2002). A note on the single machine serial batching scheduling problem to minimize maximum lateness with precedence constraints. *Operations Research Letters, 30,* 66 – 68.

Wall, M. (2005). *GAlib Documentation.* Retrieved April 15, 2006, form Massachusetts Institute of Technology Web site: http://lancet.mit.edu/galib-2.4/GAlib.html

VITA

Graduate College
University of Nevada, Las Vegas

Lewis A. Raymond

Home Address:
2120 Farmouth Circle
North Las Vegas, Nevada 89032

Degrees:
Bachelor of Science, Computer Science, 2003
University of Nevada, Las Vegas

Thesis Title: A Heuristic for Batching Jobs Under Weighted Average Completion Time

Thesis Examination Committee:
Chairperson, Dr. Wolfgang Bein, Ph. D.
Committee Member, Dr. John Minor, Ph. D.
Committee Member, Dr. Laxmi Gewali, Ph. D.
Graduate Faculty Representative, Dr. Henry Selvaraj, Ph. D.