Flow shop scheduling with two machines

V. L. Kumar Adusumilli

*University of Nevada, Las Vegas*

Follow this and additional works at: [https://digitalscholarship.unlv.edu/rtds](https://digitalscholarship.unlv.edu/rtds)

Repository Citation


This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
FLOW SHOP SCHEDULING WITH TWO MACHINES

by

V L Kumar Adusumilli

Bachelor of Technology
Nagarjuna University, India
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
August 2006
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
This Thesis prepared by Vykunta Lakshmi Kumar Adusumilli

Entitled

FLOW SHOP SCHEDULING WITH TWO MACHINES

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

Master of Science in Computer Science

By the undersigned:

Wolfgang Bein, Examination Committee Chair

Lawerence Larmore, Examination Committee Member

Laxmi Gewali, Examination Committee Member

Richard McCorkle, Graduate Faculty Representative

[Signature]

Dean of the Graduate College
ABSTRACT

Flow Shop Scheduling with Two Machines

by

V L Kumar Adusumilli

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

A flow shop problem has \( n \) jobs \((i = 1, \ldots, n)\) on \( m \) machines \((j = 1, \ldots, m)\) and a job consists two operations and the \( j^{th} \) operation of each job must be processed on machine \( j \). Any job can start only on machine \( j \) if it is completed on machine \( j - 1 \) and if machine \( j \) is free. Each operation has a known processing time \( p_{ij} \). The work here focuses on the case \( m = 2 \) where the objective is to minimize (1) the makespan \( (C_{\text{max}}) \) and (2) the average completion time \((\sum C_i)\).

We first review an efficient greedy algorithm by Johnson for \( C_{\text{max}} \) and give detailed proofs.

The we note that in the case of \( \sum C_i \) the problem is harder, in fact it is \( \text{NP}-\text{hard} \). To tackle this problem we have implemented a branch and bound algorithm to find the optimal schedules in some cases. We also constructed a genetic algorithm under MIT’s GALib C++ package. Solutions from the branch and bound algorithm are used as benchmarks for the solutions found by the genetic algorithm.
TABLE OF CONTENTS

ABSTRACT ................................................................. iii

LIST OF TABLES ......................................................... vi

LIST OF FIGURES ....................................................... vii

ACKNOWLEDGMENTS ................................................. viii

CHAPTER 1 INTRODUCTION ................................. 1
  1.1 Background .................................................. 1
  1.2 Outline of Thesis ........................................ 5
  1.3 Flow Shop Problems in General ................... 5
  1.4 Flow Shops for Two Machines ...................... 7

CHAPTER 2 FLOW SHOP PROBLEM WITH MAKESPAN MINIMIZATION ............ 10
  2.1 MINIMIZING MAKESPAN ................................ 10

CHAPTER 3 FLOW SHOP WITH AVERAGE COMPLETION TIME, BRANCH AND BOUND ...................... 16
  3.1 Branch-and-Bound Algorithm ...................... 17

CHAPTER 4 INTRODUCTION TO GENETIC ALGORITHMS ...................... 23
  4.1 Definition .................................................. 23
  4.2 Overview .................................................. 24
    4.2.1 Search Space ....................................... 25
  4.3 Implementation ......................................... 26
    4.3.1 Selection Operator ............................... 26
    4.3.2 Crossover Operator ............................. 27
    4.3.3 Mutation Operator .............................. 27
  4.4 Structure of Genetic Algorithm ................... 28
  4.5 Matthew Wall's GAlib ................................. 29
    4.5.1 General Overview ............................... 31

CHAPTER 5 RESULTS .................................................. 33

CHAPTER 6 CONCLUSION ........................................... 39

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF TABLES

1.1 Three machine flow shop ................................................................. 6
1.2 Polynomially solvable flow shop problem ....................................... 7
1.3 $NP$-hard flow shop problems ............................................................. 8

3.1 Example flow shop ................................................................................ 17

5.1 Random machine 1 and machine 2 values ........................................... 36
5.2 $n=20$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 36
5.3 Random machine 1 and machine 2 values .......................................... 37
5.4 $n=15$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 37
5.5 Random machine 1 and machine 2 values .......................................... 37
5.6 $n=10$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 38
5.7 Random machine 1 and machine 2 values .......................................... 38
5.8 $n=7$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 38
5.9 Random machine 1 and machine 2 values .......................................... 38
5.10 $n=5$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 38
5.11 Increasing machine 1 and decreasing machine 2 values ................. 39
5.12 $n=10$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 39
5.13 Decreasing machine 1 and Increasing machine 2 values ............... 39
5.14 $n=10$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 40
5.15 Minimum machine 1 values ............................................................... 40
5.16 $n=10$, Johnson’s, Branch and Bound and Genetic Algorithm .......... 40
# LIST OF FIGURES

1.1 Flow Shop for 3 machines, Case(i) .............................................................. 6
1.2 Flow Shop for 3 machines, Case(ii) ........................................................... 7
1.3 Optimal schedule upto $k$ jobs ................................................................. 9
1.4 job $j$ shifted to position immediately after job $i$ ......................................... 9

2.1 Example $F2||C_{max}$ algorithm ................................................................. 11
2.2 $F2||C_{max}$ schedule ................................................................................ 12
2.3 case (i), slack between $i$ and $j$ ............................................................... 13
2.4 case (ii), no slack between $i$ and $j$ ......................................................... 14
2.5 case (iii), slack due to $x$ ....................................................................... 14

3.1 $n = 4$, branch and bound tree after pruning .......................................... 20
3.2 Calculation of $S_1$ .................................................................................. 22
3.3 Calculation of $S_2$ .................................................................................. 23

4.1 Gene, Chromosome .................................................................................. 27
4.2 Crossover example ................................................................................... 29
4.3 Mutation example ................................................................................... 29
ACKNOWLEDGMENTS

I am grateful to my thesis advisor, Dr. Wolfgang Bein for chairing my committee and advising this work. I am indebted to him for his whole-hearted support, encouragement, enthusiasm, and inspiration throughout my graduate study, and most importantly, his faith and belief in me.

I am also grateful to Dr. Lawrence L. Larmore, Dr. Laxmi P. Gewali, and Dr. Richard McCorkle for their participation in my committee.

This thesis is dedicated to my parents for their love, faith, and support. I am very fortunate to have a wonderful set of friends, all of whom made my study at UNLV enjoyable and memorable.
CHAPTER 1

INTRODUCTION

1.1 Background

Graham et al.[12] introduced the $\alpha|\beta|\gamma$ notation to classify scheduling problems. The $\alpha$ field describes the machine environment and contains a single entry. The $\beta$ field provides details of job characteristics and scheduling constraints. This field may contain multiple entries or no entry at all. The $\gamma$ field contains the objective function to optimize. It usually contains a single entry.

In all scheduling problems, the number of jobs are denoted by $n$ and machines by $m$. Usually the subscript $i$ refers to a machine and subscript $j$ refers to a job. The following are associated with job $j$:

- **Processing Time** ($p_{ij}$): The subscript $i$ is omitted if job $j$ is processed on single machine. $p_{ij}$ represents the processing time of job $j$ on machine $i$.

- **Release Time** ($r_j$): It is the earliest time at which job $j$ can start its processing.

- **Due Date** ($d_j$): It represents the date the job is expected to complete. Completion of job after the due date is allowed, but it incurs some cost.
Deadline\(d_j\): job \(j\) must obey the deadline i.e., job \(j\) must be completed by deadline.

Weight\(w_j\): It represents the importance of job \(j\).

The possible entries for \(\alpha\) field are as follows:

- **Single Machine**\((1)\): There is only one machine in the system.

- **Parallel and Identical Machines**\((P_m)\): There are \(m\) identical machines in parallel. Each job \(j\) requires a single operation and may be processed on any one of the \(m\) machines.

- **Uniform Machines**\((Q_m)\): There are \(m\) machines in parallel, but they have different speeds. Machine \(i\), \(1 \leq i \leq m\), has speed \(s_j\). The time \(p_{ij}\) that job \(j\) spends on machine \(i\) is equal to \(p_j/s_i\), assuming that job \(j\) is completely processed on machine \(i\).

- **Unrelated machines**\((R_m)\): There are \(m\) machines in parallel, but they have different speeds. Machine \(i\), \(1 \leq i \leq m\), has speed \(s_{ij}\). The time \(p_{ij}\) that job \(j\) spends on machine \(i\) is equal to \(p_j/s_{ij}\), assuming that job \(j\) is completely processed on machine \(i\).

- **Job Shop**\((J_m)\): Each job has its own predetermined route to follow. It may visit some machines more than once and it may not visit some machines at all.

- **Flow Shop**\((F_m)\): The machines are linearly ordered and all jobs follow the same route from first to last machine.
• **Open Shop** ($O_m$): Each job needs to be processed exactly once on each of the machines, but the order of processing doesn’t matter.

The job characteristics and scheduling constraints specified in the $\beta$ field may contain multiple entries. The possible entries are $\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7, \beta_8$.

• **Preemptions** ($pmtn$): Jobs can be preempted and later resumed possibly on a different machine.

• **No-Wait** ($nwt$): The no-wait constraint is for flow shops only. Jobs are not allowed to wait between two successive machines.

• **Precedence Constraints** ($prec$): The precedence constraints specify the scheduling constraints of the jobs, in the sense that certain jobs must be completed before certain other jobs can start processing. The general form of precedence constraints, denoted by $prec$, is represented by acyclic graph, where each vertex represents a job and job $i$ precedes job $j$ if there is a directed arc from $i$ to $j$. If each job has at most one predecessor and at most one successor, the constraints are referred to as **chains**. If each job has at most one successor, the constraints are referred to as **intree**. If each job has at most one predecessor, the constraints are referred to as **outtree**.

• **Release Dates** ($r_j$): The release date $r_j$ of job $j$ is the earliest time at which job $j$ begin processing.

• **Restrictions on Number of Jobs** ($nbr$): If this symbol is present, then the
number of jobs is restricted. If this symbol is not present, then the number of jobs is unrestricted and is given as an input parameter $n$.

- **Restrictions on number of Operations in Jobs**($n_j$): This subfield is only applicable to job shops. If this symbol is present, then the number of operations of each job is restricted. If not, number of operations is unrestricted.

- **Restrictions on the Processing Times**($p_j$): If this symbol is present then the processing time of each job is restricted. If not, processing time is not restricted.

- **Deadlines**($d_j$): If the symbol is present, then the jobs are subject to deadline constraints.

  The *lateness* of the job is defined as $L_j = C_j - d_j$.

  The *tardiness* is defined as $T_j = \max(L_j, 0)$.

  The *unit penalty* of job $j$ is defined as $U_j = 1$ if $C_j > d_j$; otherwise, $U_j = 0$.

The objective function to be minimized as follows:

- **Makespan**($C_{\text{max}}$): The makespan is defined as $\max(C_1, \ldots, C_n)$.

- **Maximum Lateness**($L_{\text{max}}$): The maximum lateness is defined as $\max(L_1, \ldots, L_n)$.

- **Total Weighted Completion Time**($\sum w_j C_j$): The total (unweighted) completion time is denoted by $\sum C_j$.

- **Total Weighted Tardiness**($\sum w_j T_j$): The total (unweighted) tardiness is denoted by $\sum T_j$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
• **Weighted Number of Tardy Jobs** ($\sum w_j U_j$): The total (unweighted) number of tardy jobs is denoted by $\sum U_j$.

1.2 Outline of Thesis

Next we give formal definitions for the flow shop problem. We survey known results in the literature. Later in this chapter we focus on the case of $m = 2$. Specifically, we show that for $C_{max}$ as well $\sum C_i$ for $m = 2$ the jobs can be processed in the same order on each machine. This makes it possible to describe a schedule succinctly by just a permutation.

Chapter 2 deals with the $F2||C_{max}$ problem and contains a proof of correctness of Johnson’s $O(n \log n)$ algorithm. The $F2||\sum C_i$ problem is much harder, in fact it is NP-hard. In Chapter 3 we develop a branch and bound scheme to find optimal solutions in some cases. In this thesis a genetic algorithm is employed to solve the heuristically. The results from the branch and bound algorithm are used as benchmarks for this heuristic. Chapter 4 gives a brief overview over the genetic algorithm approach. It also contains a description of MIT’s GALib C++ package [13] which is used to implement the genetic algorithm. Chapter 5 gives results and Chapter 6 gives conclusions we have drawn from our study.

1.3 Flow Shop Problems in General

The formal definition of the flow shop is

• each job $i$ consists of $m$ operations $O_{ij}$ with the processing times $p_{ij}$ (\(j =
1, \cdots, m) where \( O_{ij} \) must be processed on machine \( M_j \), and

- there are precedence constraints of the form \( O_{ij} \rightarrow O_{i,j+1} \) \( (i = 1, \cdots, m - 1) \) for each \( i = 1, \cdots, n \), i.e. each job is first processed on machine 1, then on machine 2, then on machine 3, etc.

An excellent paper is [11].

Consider an example of flow shop with three machines with the following data.

Table 1.1

<table>
<thead>
<tr>
<th>Job</th>
<th>( p_{1i} )</th>
<th>( p_{2i} )</th>
<th>( p_{3i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>J2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>J3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Figure 1.1: Flow Shop for 3 machines, Case(i)](image)

Note that in these example figure 1.1 and figure 1.2, the order of the jobs differs across machines. For the case (i), we have \( C_{max} = 9 \) and \( \sum C_i = 18 \) In case (ii), \( C_{max} = 8 \) and \( \sum C_i = 21 \). Note that \( \sum C_i \) is better than \( C_{max} \) in case (i) whereas it
is the opposite in case (ii). The example suggests that things very much depend on the objective function.

![Figure 1.2: Flow Shop for 3 machines, Case(ii)]

We now review a number of classical results. The following table taken from [14] gives an overview of flow shop problems that are polynomially solvable (Table 1.2) and problems that are \( NP \)-hard (Table 1.3). Note that the "hardest" polynomial and the "easiest" NP-hard problems are shown.

**Table 1.2: Polynomially solvable flow shop problem**

<table>
<thead>
<tr>
<th>Problem Description</th>
<th>Author(s)</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F[</td>
<td>p_{ij} = 1; outtree; r_i</td>
<td>C_{max} )</td>
</tr>
<tr>
<td>( F[</td>
<td>p_{ij} = 1; tree</td>
<td>C_{max} )</td>
</tr>
<tr>
<td>( F2</td>
<td></td>
<td>C_{max} )</td>
</tr>
<tr>
<td>( F2</td>
<td>pmtn</td>
<td>C_{max} )</td>
</tr>
<tr>
<td>( F[</td>
<td>p_{ij} = 1; intree</td>
<td>L_{max} )</td>
</tr>
<tr>
<td>( F2[</td>
<td>p_{ij} = 1; prec; r_i</td>
<td>L_{max} )</td>
</tr>
<tr>
<td>( F[</td>
<td>p_{ij} = 1; outtree; r_i</td>
<td>\sum C_i )</td>
</tr>
<tr>
<td>( F2[</td>
<td>p_{ij} = 1; prec</td>
<td>\sum C_i )</td>
</tr>
<tr>
<td>( F[</td>
<td>p_{ij} = 1; r_i</td>
<td>\sum w_i U_i )</td>
</tr>
<tr>
<td>( F[</td>
<td>p_{ij} = 1; r_i</td>
<td>\sum w_i T_i )</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 1.3: NP-hard flow shop problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F[p_{ij} = 1; \text{intree}; r_i</td>
<td>C_{max}]$</td>
</tr>
<tr>
<td>$F[p_{ij} = 1; \text{prec}</td>
<td>C_{max}]$</td>
</tr>
<tr>
<td>$F2[	ext{chains}</td>
<td>C_{max}]$</td>
</tr>
<tr>
<td>$F2[	ext{pmtn}</td>
<td>C_{max}]$</td>
</tr>
<tr>
<td>$F2[r_i</td>
<td>C_{max}]$</td>
</tr>
<tr>
<td>$F3[	ext{pmtn}</td>
<td>C_{max}]$</td>
</tr>
<tr>
<td>$F[p_{ij} = 1; \text{outtree}; r_i</td>
<td>L_{max}]$</td>
</tr>
<tr>
<td>$F2[</td>
<td>L_{max}]$</td>
</tr>
<tr>
<td>$F2[\text{pmtn}</td>
<td>L_{max}]$</td>
</tr>
<tr>
<td>$F2[\sum C_i]$</td>
<td>Garey et al. [4]</td>
</tr>
<tr>
<td>$F2[\sum C_i]$</td>
<td>Du &amp; Leung [3]</td>
</tr>
<tr>
<td>$F_m[p_{ij} = 1; \text{chains}</td>
<td>\sum w_i C_i]$</td>
</tr>
<tr>
<td>$F_m[p_{ij} = 1; \text{chains}</td>
<td>\sum U_i]$</td>
</tr>
<tr>
<td>for each $m \geq 2$</td>
<td></td>
</tr>
<tr>
<td>$F_m[p_{ij} = 1; \text{chains}</td>
<td>\sum T_i]$</td>
</tr>
<tr>
<td>for each $m \geq 2$</td>
<td></td>
</tr>
</tbody>
</table>

1.4 Flow Shops for Two Machines

To minimize the makespan for $F2||C_{max}$ and to find the average completion time for $F2||\sum C_i$ we may restrict our attention to schedules in which the job sequence on both machines is the same. This is called permutation flow shop problem. This is an immediate consequence of the following lemma.

**Lemma 1.4.1** For the flow-shop problem $F2||\sum C_i$ and $F2||C_{max}$ there exists an optimal schedule in which both machines process the jobs in the same order.

**Proof:** Consider an optimal schedule in which the processing order on both machines is identical for the first $k$ scheduled jobs, where $k < n$ is maximal. Let $i$ be
the $k$-th job and let $j$ be the job scheduled on machine 2 after the second operation of job $i$. Then we may have a situation as shown in Figure 1.3.

![Figure 1.3: Optimal schedule upto $k$ jobs](image)

If on machine 1 we shift job $j$ to the position immediately after job $i$ and move the jobs scheduled previously between job $i$ and job $j$ by $p_{ij}$ time units to the right, we get another optimal schedule. This contradicts the maximality of $k$. (figure 1.4)
CHAPTER 2

FLOW SHOP PROBLEM WITH MAKESPAN MINIMIZATION

2.1 Minimizing Makespan

Problem F2||C_{max} is the only flow shop problem with C_{max} criterion which is polynomially solvable if the processing times are arbitrary. The Johnson's Algorithm is used to minimize the makespan.

**Johnson's Algorithm:** This algorithm is used to minimize the makespan of all jobs such that if all the jobs are scheduled in L : L(1),...,L(n) this order on both machines. At any moment consider the job i that has the smallest value of \( p_{i1} \) or \( p_{i2} \) where \( i = 1,...,n \). We construct \( L : L(1),...,L(n) \) by concatenating \( T : L(1),...,L(t) \) and \( R : L(t+1),...,L(n) \).

Initially we have \( X = \{1,...,i,...,n\} \), be the set of all jobs that are not scheduled yet. We will do the following until \( X \) gets empty.

For all jobs in \( X \), find the job that has the smallest processing time (\( p_{i1} \) or \( p_{i2} \)). From that if job \( i \) has smallest \( p_{i1} \) value then job \( i \) is added to the tail of \( T \) i.e, \( T \circ i \) and if otherwise job \( i \) is added to the front of \( R \) i.e, \( i \circ R \).

After all the jobs have been scheduled in \( T \) and \( R \), then we get the order \( L : L(1),...,L(n) \) by concatenating \( T \) and \( R \).
Algorithm 1 $F2||C_{max}$

1. $X := \{1, \ldots, n\}; T := \emptyset; R := \emptyset$
2. While $X \neq \emptyset$ DO
   BEGIN
3. Find job $i$ that has smallest $p_{i1}$ or $p_{i2}$
4. IF $p_{i1}$ is small THEN $T := T \circ i^*$ ELSE $R := i^* \circ R$
5. $X := X \setminus \{i^*\}$
   END;
6. $L := T \circ R$

The example in the figure 2.1 shows how Algorithm $F2||C_{max}$ works.

![Figure 2.1: Example $F2||C_{max}$ algorithm](image)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
To prove that Algorithm $F_2||C_{\text{max}}$ is correct we need the following two lemmas.

**Lemma 2.1.1** Let $L : L(1), \ldots, L(n)$ be a list constructed by Algorithm $F_2||C_{\text{max}}$ then

$$\min\{p_{i1}, p_{j2}\} < \min\{p_{j1}, p_{i2}\}$$

implies that job $i$ appears before job $j$ in $L$.

**Proof:**

- Give that $\min\{p_{i1}, p_{j2}\} < \min\{p_{j1}, p_{i2}\}$ If we have, $p_{i1}$ as the smaller value on the left i.e, $p_{i1} < \min\{p_{i2}, p_{j1}\}$ then $p_{i1} < p_{i2}$ implies that job $i$ belongs to $T$.

  At some point, if job $j$ is added to $R$ then we are done. Otherwise we have $p_{i1} < p_{j1}$ in which job $i$ appears after $j$ in $T$.

- If $p_{j2}$ has the smaller value i.e, $p_{j2} < \min\{p_{i2}, p_{j1}\}$ then $p_{j2} < p_{j1}$ implies that job $j$ belongs to $R$. 

Figure 2.2: $F_2||C_{\text{max}}$ schedule
At some point, if job $i$ is added to $T$ then we have finished. Otherwise we have $p_{i2} < p_{i2}$ in which job $i$ appears after $j$ in $R$.

**Lemma 2.1.2** Consider a schedule in which job $j$ is scheduled immediately after job $i$. Then

$$\min\{p_{j1}, p_{j2}\} \leq \min\{p_{i1}, p_{j2}\}$$  \hspace{1cm} (2.1)

implies that $i$ and $j$ can be swapped without increasing the $C_{\text{max}}$ value.

**Proof:** If $j$ is scheduled immediately after $i$, then we have three possible cases which are shown in figure. Denote by $w_{ij}$ the length of the time period from the start of job $i$ to the finishing time of job $j$ in this situation, we have

**Case (i):**

\[
\begin{array}{|c|c|c|}
\hline
M1 & \ldots & i & j & \ldots \\
\hline
M2 & i & j & \\
\hline
\end{array}
\]

Figure 2.3: case (i), slack between $i$ and $j$

The figure 2.3 shows a slack between $i$ and $j$ which is due to job $j$. Then $w_{ij}$ for this case is

$$w_{ij} = p_{i1} + p_{j1} + p_{j2}$$
Case (ii):

\[
\begin{array}{ccc}
M1 & i & j \\
\ldots & \ldots & \\
M2 & i & j \\
\end{array}
\]

Figure 2.4: case (ii), no slack between \(i\) and \(j\)

The figure 2.4 shows a no slack between \(i\) and \(j\) which is due to job \(i\). Then \(w_{ij}\) for this case is

\[w_{ij} = p_{i1} + p_{i2} + p_{j2}\]

Case (iii):

\[
\begin{array}{ccc}
M1 & i & j \\
\ldots & \ldots & \\
M2 & i & j \\
\end{array}
\]

\(x\)

Figure 2.5: case (iii), slack due to \(x\)

The figure 2.5 shows a slack due to \(x\). Then \(w_{ij}\) for this case is

\[w_{ij} = x + p_{i2} + p_{j2}\]

From case(i), case(ii), case(iii)

\[w_{ij} = \max\{p_{i1} + p_{j1} + p_{j2}, p_{i1} + p_{i2} + p_{j2}, x + p_{i2} + p_{j2}\}\]
Similarly, we have

\[ w_{ij} = \max \{ p_{j1} + p_{i2} + \max \{ p_{i1}, p_{i2} \}, x + p_{i2} + p_{j2} \} \]

if \( i \) is scheduled immediately after \( j \).

\[ \max \{-p_{i1}, -p_{j2}\} \leq \max \{-p_{j1}, -p_{i2}\}. \]

Adding \( p_{i1} + p_{i2} + p_{j1} + p_{j2} \) to the both sides of this inequality, we get

\[ p_{j1} + p_{i2} + \max \{ p_{i1}, p_{j2} \} \leq p_{i1} + p_{j2} + \max \{ p_{j1}, p_{i2} \}, \]

which implies \( w_{ji} \leq w_{ij} \). Thus, swapping \( i \) and \( j \) will not increase the \( C_{\text{max}} \) value.

Now it is not difficult to prove the correctness of Algorithm \( F2||C_{\text{max}} \).

**Theorem 2.1.3** The sequence \( L : L(1), \ldots, L(n) \) constructed by Algorithm \( F2||C_{\text{max}} \) is optimal.

**Proof by contradiction:** Assume that the sequence \( L : L(1), \ldots, L(n) \) is not an optimal sequence. Then we consider an optimal schedule \( S \), that corresponds to \( L \) as much as possible.

\[ L(v) = S(v) \text{ for } v = 1, \ldots, s - 1 \]

\[ s - 1 \leq L(s) \neq S(s) =: j \]

where \( s \) is maximal. Then job \( i \) is a (not necessarily immediate) successor of \( j \) in \( S \).

Let \( k \) be a job scheduled between job \( j \) and job \( i \) or \( k = j \) in \( S \). In \( L \), job \( k \) is scheduled after job \( i \). Thus, by Lemma 2.1.1 we must have

\[ \min \{ p_{k1}, p_{i2} \} \geq \min \{ p_{i1}, p_{k2} \} \quad (2.2) \]
(otherwise job i would follow job k in L).

The above inequality holds for each such job k. Applying Lemma 2.1.2 to R, we may swap each immediate predecessor k of job i with i without increasing the objective value. We finally get a sequence $S^-$ with $S^-(v) = L(v)$ for $v = 1, \ldots, s$ which contradicts the maximality of s.
CHAPTER 3

FLOW SHOP WITH AVERAGE COMPLETION TIME, BRANCH AND BOUND

We consider the problem $F2||\sum C_i$ with $n$ jobs $i = 1, \ldots, n$. We first show that the schedule produced by Johnson's algorithm can be arbitrarily bad for weighted average completion $\sum C_i$. To see this, consider an example flowshop that has $n$ jobs in which $\epsilon$ is considered very small and $k$ very large. We refer to the $n^{th}$ job as the "large" job.

<table>
<thead>
<tr>
<th>job</th>
<th>$p_{i1}$</th>
<th>$p_{i2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>2</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>$\epsilon/2$</td>
<td>$k$</td>
</tr>
</tbody>
</table>

Table 3.1: Example flow shop

For the data, it is obvious that the optimal schedule for $\sum C_i$ would schedule the large job last. Then we have for $\sum C_i$:

$$\sum C_i = C_1 + C_2 + \ldots + C_n$$

$$= 2\epsilon + 3\epsilon + \ldots + n\epsilon + (n\epsilon + \epsilon/2 + k)$$

$$= \text{lower order terms} + k$$

Now applying Johnson's algorithm, it would schedule the large job first. Now we
calculate $\sum C_i$:

$$\sum C_i = C_1 + C_2 + \ldots + C_n$$

$$= (\epsilon/2 + k) + (C_1 + (\epsilon)) + \ldots + (C_1 + (\epsilon + \epsilon) + \ldots (C_1 + (\epsilon + \ldots + \epsilon))$$

$$= nk + \text{lower order terms}$$

From the two examples we can see that Johnson's Algorithm is arbitrarily bad if $n$ is arbitrarily large.

3.1 Branch-and-Bound Algorithm

Branch-and-bound is one of the methods for solving combinatorial optimization problems. It is based on the idea of intelligently enumerating all feasible solutions.

We assume that a discrete optimization problem $P$ to be solved is a minimization problem. We also consider that subproblems of $P$ which are defined by a subsets $S'$ of the set $S$ of feasible solution of $P$. It is convenient to identify $P$ and its subproblems with the corresponding subset $S' \subseteq S$. Two things are needed for a branch-and-bound algorithm.

1. **Branching**: $S$ is replaced by smaller problems $S_i(i = 1, \ldots, r)$ such that $\cup_{i=1}^r S_i = S$. This process is called branching. Branching is a recursive process, i.e. each $S_i$ is the basis of another branching. The whole branching process is represented by a branching tree. $S$ is the root of the branching tree, $S_i(i = 1, \ldots, r)$ are the children of $S$, etc. The discrete optimization problems created by the branching process are called subproblems.
2. **Lower Bounding:** An algorithm is available for calculating a lower bound for the objective values of all feasible solutions of a subproblem.

3. **Upper Bounding:** We calculate an upper bound $upperBound$ of the objective value of $P$. The objective value of any feasible solution will provide such an upper bound. If the lower bound of a subproblem is greater than or equal to $upperBound$, then this subproblem cannot yield a better solution. Thus, we need not continue to branch from the corresponding node in the branching tree. To stop the branching process in many nodes of the branching tree, the bound $upperBound$ should be as small as possible. Therefore, at the beginning of the branch-and-bound algorithm we apply some heuristic to find a good feasible solution with small value $upperBound$. After branching many times we may reach a situation in which the subproblem has only one feasible solution. Then the lower bound $lowerBound$ of the subproblem is set equal to the objective value of this solution and we replace $upperBound$ by $lowerBound$ if $lowerBound < upperBound$.

Algorithm Branch-and-Bound summarizes these basic ideas.
Algorithm 2 Branch-and-Bound Algorithm

1. lowerBound, upperBound = feasiblesolution GENERATE_NODES(a, i)
2. IF i = n THEN currentSolution = calsch(n, a) END IF
3. IF currentSolution < upperBound THEN UPDATE upperBound
   ELSE
      (a) CALCULATE lowerBound
      (b) IF lowerBound ≥ upperBound THEN CUT
           ELSE
              i. FOR i + 1 TO n DO
                 BEGIN
              ii. SWAP
              iii. CALL GENERATE_NODES(a, i + 1) END FOR
           END IF
   END IF

Consider the number of jobs to be 4. The following possibilities are represented as tree as shown in figure 3.1.

![Branch and Bound Tree](image)

Figure 3.1: $n = 4$, branch and bound tree after pruning
By applying the algorithm we can prune the tree. There are two possibilities in doing this:

1. when we are at middle of the tree, if the lowerBound calculated is greater than the upperBound then we will not proceed to the next node.

2. when we are at leaf node, if the currentSolution calculated is better than the upperBound the we update upperBound with currentSolution.

We are now ready to design the scheme for the flow shop problem:

Suppose we are at node at which the jobs in the set \( M \subseteq \{1, \cdots , n\} \) have been scheduled, where \( |M| = r \). Let \( i_k, k = 1, \cdots , n \) be the index of the \( k \)-th job under any schedule which is a descendant of the node under construction.

Consider the number of jobs to be 4. The possibilities are represented as tree as shown in figure. For example the node (23) represents the fact that jobs 2 and 3 are fixed in this order and jobs 1 and 4 could still be in any order after jobs 2,3. The cost of this schedule, which we wish to bound, is

\[
S = \sum_{i \in M} C_i + \sum_{i \not\in M} C_i
\]  

For the second sum in (1) we will derive two possible lower bounds.

1. If every job \( i \not\in M \) could start its processing on machine 2 immediately after completing its processing on machine 1, the second sum in (1) would become

\[
S_1 = \sum_{k=r+1}^{n} \left[ \sum_{i \in M} p_{i1} + (n - k + 1)p_{i1} + p_{i2} \right]
\]  

(3.2)
Calculation of $S_1$: Consider the jobs $r+1, \ldots, n$ on machine 1 are done without any delay.

$$S_1 = C_{r+1} + C_{r+2} + \cdots + C_k + \cdots + C_n \quad (3.3)$$

$$C_{r+1} = p_{ir+11} + p_{ir+12} + \sum_{i \in M} p_{il}$$

$$C_{r+2} = p_{ir+11} + p_{ir+21} + p_{ir+22} + \sum_{i \in M} p_{il}$$

$$\vdots$$

$$C_k = p_{ir+11} + p_{ir+21} + \cdots + p_{ik1} + p_{ik2} + \sum_{i \in M} p_{il}$$

$$= (n - k + 1)p_{ik1} + p_{ik2} + \sum_{i \in M} p_{il}$$

$$\vdots$$

$$C_n = p_{ir+11} + p_{ir+21} + \cdots + p_{in1} + \cdots + p_{in1} + p_{in2} + \sum_{i \in M} p_{il}$$
From the above eqns, it is achieved that we need to take values in the increasing order of $p_{i1}$. Therefore from eqn (3), we obtain

$$S_1 = \sum_{k=r+1}^{n} \left[ \sum_{i \in M} p_i + (n - k + 1)p_{ki1} + p_{ki2} \right]$$

2. $\max\{C_{ir}, \sum_{i \in M} p_{i1} + \min_{i \notin M} p_{i1}\}$ is a lower bound on the start of the first job $i \notin M$ on machine 2. Thus the second sum in (1) would be bounded by

$$S_2 = \sum_{k=r+1}^{n} \left[ \max\{C_{ir}, \sum_{i \in M} p_i + \min_{i \notin M} p_{i1}\} + (n - k + 1)p_{i1}\right]. \quad (3.4)$$

Calculation of $S_2$: Consider the jobs $r+1, \ldots, n$ on machine 2 are done without any delay.

$$S_2 = C_{r+1} + C_{r+2} + \cdots + C_k + \cdots + C_n \quad (3.5)$$

$$C_{r+1} = p_{r+1} + \max\{C_{ir}, \sum_{i \in M} p_i + \min_{i \notin M} p_{i1}\}$$

$$C_{r+2} = p_{r+2} + p_{r+2} + \max\{C_{ir}, \sum_{i \in M} p_i + \min_{i \notin M} p_{i1}\}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
\[ C_k = p_{i_k} + \cdots + p_{i_{k+2}} + p_{i_{r+2}} + \max\{C_{i_r}, \sum_{i \in M} p_{i_1} + \min_{i \in M} p_{i_1}\} \]

\[ C_n = p_{i_n} + \cdots + p_{i_{k_2}} + \cdots + p_{i_{r+2}} + p_{i_{r+1}2} + \max\{C_{i_r}, \sum_{i \in M} p_{i_1} + \min_{i \in M} p_{i_1}\} \]

From the above eqns, it is achieved that we need to take values in the increasing order of \( p_{i_2} \). Therefore from eqn(3), we obtain

\[ S_2 = \sum_{k=r+1}^{n} [\max\{C_{i_r}, \sum_{i \in M} p_{i_1} + \min_{i \in M} p_{i_1}\} + (n - k + 1)p_{i_{k+2}}]. \]

Combining the two bounds we get

\[ \sum_{i \in M} C_i + \max\{S^*_1, S^*_2\} \quad (3.6) \]

which is the computed lower bound.
CHAPTER 4

INTRODUCTION TO GENETIC ALGORITHMS

Genetic Algorithms were first invented in the early 1970s by John Holland at University of Michigan to imitate some of the processes observed in natural selection. The main idea behind GA is to solve optimization problems.

4.1 Definition

Genetic Algorithms (GAs) are a class of adaptive heuristic search technique based on the biological process of natural selection and genetics. They intelligently exploit the use of random search to solve optimization problems. Although randomized, GAs are by no means random, within the search space they exploit historical information to direct the search into the region of better performance. The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, specially those follow the principles first laid down by Charles Darwin of "survival of the fittest." Since in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

GAs are more robust than the conventional AI. Unlike older AI systems, they do not break easily even if the inputs changed slightly, or in the presence of reasonable
noise. Also, in searching a larger state-space, or $n$-dimensional surface, a genetic algorithm may offer significant benefits over more typical search of optimization techniques such as linear programming, heuristic, depth-first, breath-first, and praxis).

4.2 Overview

GAs simulate the survival of the fittest among individuals over consecutive generations for solving a problem. Each generation consists of a population of character strings that are analogous to the chromosome that we see in our DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution.

GAs are based on an analogy with the generic structure and behavior of chromosomes within a population of individuals using the following foundations:

- Individuals in a population compete for resources and mates.

- Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.

- Genes from 'good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.

- Thus each successive generation will become more suited to their environment.
4.2.1 Search Space

A population of individuals are maintained within a search space for a GA, each representing a possible solution to a given problem. Each individual is coded as a finite length vector of components, or variables, in terms of some alphabets, usually the binary alphabet \{0, 1\}. To continue the genetic analogy these individuals are linked to chromosomes and the variables are analogous to genes. Thus a chromosome (solution) is composed of several genes (variables). A **fitness score** is assigned to each solution representing the abilities of an individual to 'compete'. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective 'breeding' of the solutions to produce 'offspring' better than the parents by combining information from the chromosomes.

![Gene and Chromosome Diagram](image)

Figure 4.1: Gene, Chromosome

The GA maintains a population of \( n \) chromosomes (solutions) with associated fitness values. Parents are selected to mate, on the basis of their fitness, producing offspring via a reproductive plan. Consequently highly fit solutions are given more opportunities to reproduce, so that offspring inherit characteristics from each parent. As parents mate and produce offspring, room must be made for the new arrivals since the population is kept at a static size. Individuals in the population die and are...
replaced by the new solutions, eventually creating a new generation once all mating opportunities in the old population have been exhausted. In this way it is hoped that over successive generations better solutions will thrive while the least fit solutions dies out.

New generations of solutions are produced containing, on average, more good genes than a typical solution in a previous generation. Each successive generation will contain good 'partial solutions' than previous generations. Eventually, once the population has converged and is not producing offspring noticeably different from those in previous generations, the algorithm itself is said to have converged to a set of solutions to the problem at hand.

4.3 Implementation

Based on natural selection, after an initial population is randomly generated, the algorithm evolves through three operators

1. selection which equates to survival of fittest.

2. crossover which represents mating between individuals.

3. mutation which introduces random modifications.

4.3.1 Selection Operator

Preference is given to better individuals, allowing them to pass on their genes to the next generation. The goodness of each individual depends on its fitness. Fitness may be determined by an objective function or by a subjective judgment.
4.3.2 Crossover Operator

This is the prime distinguished factor of GA from other optimization techniques. Two individuals are chosen from the population using the selection operator. A crossover site along the bit strings is randomly chosen. The values of the strings are exchanged up to this point. For example, if $S_1 = 000000$ and $S_2 = 111111$ and the crossover point is 2 then $S_1^1 = 110000$ and $S_2^1 = 001111$. The new offspring created from this mating are put into the next generation of the population. By recombining portions of good individuals, this process is likely to create even better individuals.

![Figure 4.2: Crossover example](image)

4.3.3 Mutation Operator

With low probability, a portion of the new individuals will have some of their bits flipped. Its purpose is to maintain diversity within the population and inhibit premature convergence. Mutation alone induces a random walk through the search space.

![Figure 4.3: Mutation example](image)
Effects of Genetic Operators

1. Using selection alone will tend to fill the population with copies of the best individual from the population.

2. Using selection and crossover operators will tend to cause the algorithms to converge on a good but sub-optimal solution.

3. Using mutation alone induces a random walk through the search space.

4. Using selection and mutation creates a parallel, noise-tolerant, hill climbing algorithm

4.4 Structure of Genetic Algorithm

1. randomly initialize population(t).

2. determine fitness of population(t).

3. repeat

   (a) select parents from population(t).

   (b) perform crossover on parents creating population(t + 1).

   (c) perform mutation of population(t + 1).

   (d) determine fitness of population(t + 1).

4. until best individual is found.
4.5 Matthew Wall's GAlib

We implemented the 2-machine flow shop problem using the Matthew Wall's GAlib. GAlib is a C++ library developed at the Massachusetts Institution of Technology designed to assist in the development of genetic algorithm applications. The library contains numerous classes that offer functionality and flexibility in the design of optimization applications with genetic algorithms. The library includes default genetic algorithm models, genome types, and genetic operators for the quick creation of simple applications, and the ability to customize GAlib for more complicated optimizations. This library was programmed so that it may be used on a variety of compilers on many platforms. The library has been used successfully on DOS/Windows, Windows NT/95, MacOS, and UNIX systems. GAlib was designed to work with Microsoft Visual C++, Borland C++ and GNU compilers, as well as others. Our applications were written under the g++ environment and all examples given in this paper were written in that environment.

GAlib supports several different models of genetic algorithms. The simple GA is the standard genetic algorithm, where after each generation, the population of possible solutions is completely replaced by the mutation and crossover of the previous generation. The incremental and steady state genetic algorithms both replace only a portion of the population with each generation. The deme GA evolves multiple population and migrates individuals from one population to another. With this GA model, GAlib can run on parallel processors, evolving each population on a separate processor. It is also possible to develop a custom GA to suit the purposes of an
application.

Each of these GA types is simple to implement and gives a great deal of freedom in their operation. A variety of algorithm termination methods, selection methods, random number generators, and statistics are available to choose from. Most of these features may be customized as well. Also, crossover and mutation probabilities, population overlap, and population size are customizable quantities.

Any datatype in C++ may be used to create a genome type. GAlib includes several of the most common genome types. These include one-dimensional, twodimensional and three-dimensional arrays of binary elements, and 1-D arrays of real or character valued elements. In addition to these, are lists, trees, 1-D, 2-D, and 3-D arrays, all of which are templates and allow the programmer to select any valid C++ data type. All the arrays may be set to any desired length, and the trees and lists have dynamic sizes. Each of these genome types has built-in initialization, crossover, mutation and comparison methods, which can be customized by the programmer. The only routine that must be coded by the programmer is the objective function. This is the function that evaluates an individual from the population and calculates a fitness score.

The versatility and ease of GAlib makes it a useful tool for implementing genetic algorithms. It is versatile enough to apply to complex optimization problems through customization, yet still simplifies the work. For simple genetic algorithm applications, little programming is required. Also, because GAlib includes a large variety of genetic algorithm and genome types and is written with a hierarchical structure, it is simple
to modify software already written with GAlib to perform new tasks.

4.5.1 General Overview

When programming using GAlib, one will work primarily with two classes: a genome class and a genetic algorithm class. A genome represents a single individual in the population of solutions. The genetic algorithm defines how the solution will be evolved. In addition to defining these two classes, an objective function is needed. GAlib supplies the two classes, but the objective function must be programmed. If the classes supplied by GAlib are inadequate to the task at hand, they may be customized, or the programmer may develop his or her own implementations.

The three necessary steps to developing an application using GAlib are to:

- define a representation
- define the genetic operators
- define the objective function

GAlib includes many examples, built-in operators, and genome representations to aid in the first two steps, but the objective function must be implemented by the programmer. Once these three steps have been completed, the genetic algorithm can begin its search for a solution.

A single object is used to represent a possible solution to an optimization problem. The genetic algorithm will create a population of this structure that is supplied. Then, the genetic algorithm will operate on the population in an attempt to evolve the best solution. The data genome structure used by GAlib is called a GAGenome. The
GAlib library contains genomes represented as an array, a list, a tree, and a string of binary bits. These genome types are derived from the class GAGenome and a data structure class. For example, the class GATreeGenome class, which represents a tree structure, inherits from the class GAGenome and the class GATree. The programmer may choose from one of these built-in genomes or if none of GAlib's available genome types will work as a representation of a solution to the problem at hand, the programmer may develop his or her own GAGenome type. The programmer must write the new type inherited from the class GAGenome and his or her own data structure class.

In addition to the genome types available, GAlib offers a selection of genetic algorithm models to choose from. The basic types of genomes included are the simple, steady-state, and incremental genetic algorithms. These GA types inherit from the class GAGeneticAlgorithm. They differ from each other in the methods that new population members are created and replace the old population members.

A properly implemented genetic algorithm will be capable of performing local searches as well as global searches for the best solution to an optimization problem. A feature of GAlib is that it is simple to modify the parameters of the genetic algorithm in order to find the best conditions for the search.
CHAPTER 5

RESULTS

The following results are developed using Genetic Algorithms and using Branch and Bound schema for two machine flow shop scheduling problem. The following assumptions are made

1. While implementing Branch and Bound, we calculate an initial feasible solution which is the sum of completion time all the processes in the ascending order.

2. While implementing Genetic Algorithms, the crossover probability and mutation probability are 0.85 and 0.01 respectively which are proved to be a good selection by experience.

The following results are obtained by applying Johnson’s algorithm, branch and bound algorithm and genetic algorithms to various type of data like arbitrary, maximum, minimum $p_{i1}$ and $p_{i2}$ values. The results obtained by genetic algorithms are close to the results obtained by branch and bound.
Table 5.1: Random machine 1 and machine 2 values

<table>
<thead>
<tr>
<th>Process value</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: n=20, Johnson’s, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th>n = 20</th>
<th>Johnson’s</th>
<th>Branch-and-Bound</th>
<th>GA gen=500,pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum C_i )</td>
<td>1247</td>
<td>1054</td>
<td>1096</td>
</tr>
<tr>
<td>( C_{\text{max}} )</td>
<td>112</td>
<td></td>
<td>1087</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1103</td>
</tr>
</tbody>
</table>
Table 5.3: Random machine 1 and machine 2 values

<table>
<thead>
<tr>
<th>Process value</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.4: \( n=15 \), Johnson’s, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th>( n = 15 )</th>
<th>Johnson’s</th>
<th>Branch-and-Bound</th>
<th>GA gen=150,pop=50</th>
<th>gen=200,pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum C_i )</td>
<td>717</td>
<td>601</td>
<td>615</td>
<td>620</td>
</tr>
<tr>
<td>( C_{max} )</td>
<td>80</td>
<td></td>
<td>621</td>
<td>628</td>
</tr>
</tbody>
</table>

Table 5.5: Random machine 1 and machine 2 values

<table>
<thead>
<tr>
<th>Process value</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 5.6: n=10, Johnson's, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th>$n = 10$</th>
<th>Johnson's</th>
<th>Branch-and-Bound</th>
<th>GA gen=150, pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum C_i$</td>
<td>331</td>
<td>292</td>
<td>297</td>
</tr>
<tr>
<td>$C_{max}$</td>
<td>54</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| | | |
| | 292 | |
| | 294 | |
| | 295 | |

Table 5.7: Random machine 1 and machine 2 values

<table>
<thead>
<tr>
<th>Process</th>
<th>value</th>
<th>Machine1</th>
<th>Machine2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: n=7, Johnson's, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th>$n = 7$</th>
<th>Johnson's</th>
<th>Branch-and-Bound</th>
<th>GA gen=150, pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum C_i$</td>
<td>182</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>$C_{max}$</td>
<td>36</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.9: Random machine 1 and machine 2 values

<table>
<thead>
<tr>
<th>Process</th>
<th>value</th>
<th>Machine1</th>
<th>Machine2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.10: n=5, Johnson's, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th>$n = 5$</th>
<th>Johnson's</th>
<th>Branch-and-Bound</th>
<th>GA gen=150, pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum C_i$</td>
<td>97</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>$C_{max}$</td>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.11: Increasing machine 1 and decreasing machine 2 values

<table>
<thead>
<tr>
<th>Process</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.12: n=10, Johnson’s, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th>n=10</th>
<th>Johnson’s</th>
<th>Branch-and-Bound</th>
<th>GA gen=200, pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΣCi</td>
<td>395</td>
<td>337</td>
<td>340</td>
</tr>
<tr>
<td>Cmax</td>
<td>56</td>
<td></td>
<td>343</td>
</tr>
</tbody>
</table>

Table 5.13: Decreasing machine 1 and Increasing machine 2 values

<table>
<thead>
<tr>
<th>Process</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
Table 5.14: n=10, Johnson’s, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Johnson’s</th>
<th>Branch-and-Bound</th>
<th>GA gen=200, pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum C_i$</td>
<td>395</td>
<td>337</td>
<td>343</td>
</tr>
<tr>
<td>$C_{max}$</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>343</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>342</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>343</td>
</tr>
</tbody>
</table>

Table 5.15: Minimum machine 1 values

<table>
<thead>
<tr>
<th>Process Value</th>
<th>Machine 1</th>
<th>Machine 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.16: n=10, Johnson’s, Branch and Bound and Genetic Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Johnson’s</th>
<th>Branch-and-Bound</th>
<th>GA gen=200, pop=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum C_i$</td>
<td>490</td>
<td>430</td>
<td>431</td>
</tr>
<tr>
<td>$C_{max}$</td>
<td>81</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>434</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>431</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>431</td>
</tr>
</tbody>
</table>
CHAPTER 6

CONCLUSION

We have implemented a branch and bound algorithm for calculating the optimal schedule and used it as a bench-mark to compare the schedule obtained by Genetic Algorithm. It shows that there is a difference of 2% between the results obtained by Genetic Algorithms and Branch-and-Bound.

The future work would be implementing the branch and bound algorithm for more number of machines. If there is a general branch and bound algorithm for all the NP hard problems then we might compare them with genetic algorithms and see how they perform. Implementing branch and bound with less time complexity.
APPENDIX A

JOHNSON'S ALGORITHM

/* First Sort all the jobs according to process values*/
QuickSort(a,1,2*n);

/* From the sorted list now we separate left an right list*/
for(int i = 1 ; i <=2*n ;i++)
{
    if(a[i].mlm2 == 1)
    {
        int flag=0;
        for(int k=1;k<=count2;k++)
            if(c[k]==a[i].process)flag=1;
        if(flag==0)
        {
            count1++;
            b[count1]= a[i].process;
        }
    }
    else

int flag=0;
for(int k=1;k<=count1;k++)
    if(b[k]==a[i].process)flag=1;
if(flag==0)
    {
        count2++;
        c[count2] = a[i].process;
    }
/* From the two lists we compute C_max*/
for (int j = 1;j <= n;j++)
    {
        cs1 = 0;cs2 = 0;
        if (cs1 == 0)
            {
                ts1 = ts1 + proc[d[j]].machine1;
                cs1 = 1;
            }
        if (cs1 == 1 && cs2 == 0)
            {

if (ts1 < ts2)
    ts2 = ts2 + proc[d[j]].machine2;
else
    ts2 = ts1 + proc[d[j]].machine2;
    cs2 = 1;
}
if (cs1 == 1 && cs2 == 1)
    temp2 = temp2 + ts2;
}
cout <<"Sum C(i):"<<temp2<<" and Cmax: " << ts2 <<endl;
/* Branch and Bound tree calculation */

void generate_nodes(vector <int> a , int i)
{
    if ( i == n) // at leaf node
    {
        /* Calculate the current solution which
           is sum of completion time of all jobs at leaf */
        currentSolution = calsch(n,a);
        if (currentSolution < upperBound)
            upperBound = currentSolution;
    }
    else
    {
        for (int j=1 ; j <= i; j++)
        {
            cs1 = 0; cs2 = 0;
            if (cs1 == 0)
{ 
    ts1 = ts1 + proc[a[j]].machine1;
    cs1 = 1;
}
if (cs1 == 1 && cs2 == 0)
{
    if (ts1 < ts2)
        ts2 = ts2 + proc[a[j]].machine2;
    else
        ts2 = ts1 + proc[a[j]].machine2;
    cs2 = 1;
}
if (cs1 == 1 && cs2 == 1)
{
    temp = temp + ts2;
    if (j == i)
        temp2 = ts2;
}
pi = pi + proc[a[j]].machine1;
}
for (int k = i+1; k <= n; k++)
{

b[a[k]].process_value = proc[a[k]].process_value;
b[a[k]].machine1 = proc[a[k]].machine1;
b[a[k]].machine2 = proc[a[k]].machine2;
}

for (int k = i+1;k <= n; k++)
{

e[a[k]].process_value = proc[a[k]].process_value;
e[a[k]].machine1 = proc[a[k]].machine1;
e[a[k]].machine2 = proc[a[k]].machine2;
}

/* Sorting the structure according to machine 1 value */
for (int l = n ; l >= i+1 ; l--)
{
    for (int j = i+2;j <= l;j++)
    {
        if( b[c[j-1]].machine1 > b[c[j]].machine1 )
        {
            temp1 =b[c[j-1]];
b[c[j-1]] = b[c[j]];
b[c[j]] = temp1;
        }
    }
}
/* Sorting according to machine 2 value */
for (int l = n ; l >= i+1 ; l--)
{
    for (int j = i+2 ; j <= l ; j++)
    {
        if (e[d[j-1]].machine2 > e[d[j]].machine2 )
        {
            temp3 = e[d[j-1]];
            e[d[j-1]] = e[d[j]];
            e[d[j]] = temp3;
        }
    }
}

/* Calculation of S1 */
for (int l = i+1;l<=n;l++)
    S1=S1+((nl+1)*proc[b[c[l]].process_value].machine1 +proc[b[c[l]].process_value].machine2)+pi;

/* Calculation of S2 */
for (int l=i+1; l<=n; l++)
{
    min = b[c[i+1]].machine1;
    sigmapi = pi+min;
    Cr = temp2;
    if (sigmapi>Cr)
        maxCrpi = sigmapi;
    else
        maxCrpi = Cr;
    S2 = S2+maxCrpi+(n-l+1)*proc[e[d[l]].process_value].machine2;
}

if (S1>S2)
    maxS1S2 = S1;
else
    maxS1S2 = S2;
lowerBound = temp + maxS1S2;

if (lowerBound >= upperBound )
{
}
else
{
    for (int k = i+1; k <= n; k++)
    {
        tmp = a[i+1];
        a[i+1] = a[k];
        a[k] = tmp;
        generate_nodes(a, i+1);
    }
}
}
APPENDIX C

GENETIC ALGORITHM

float Objective (GAGenome &g)
{
    int cs1,cs2;
    int score = 0;
    pw prowt[GenSize],newprowt[GenSize];
    GA1DArrayGenome<int>& genome = (GA1DArrayGenome<int>&)g;
    ifstream in(PROCESS_FILE);
    if(!in)
    {
        cerr << "Couldn't read data file " << PROCESS_FILE << "\n";
        exit(1);
    }
    for ( int j = 1; j <= GenSize && !in.eof(); j++)
    {
        in >> prowt[j].p;
        in >> prowt[j].m1;
        in >> prowt[j].m2;
    }
in.close();

for (int i = 1; i <= GenSize; i++)
{
    newprowt[i].p = prowt[genome.gene(i-1)].p;
    newprowt[i].m1 = prowt[genome.gene(i-1)].m1;
    newprowt[i].m2 = prowt[genome.gene(i-1)].m2;
}

int temp=0;
int ts1=0,ts2=0;

for (int j = 1; j <= GenSize; j++)
{
    cs1 = 0;cs2 = 0;
    if (cs1 == 0)
    {
        ts1 = ts1 + newprowt[j].m1;
        cs1 = 1;
    }
    if (cs1 == 1 && cs2 == 0)
    {
        if(ts1 < ts2)
            ts2 = ts2 + newprowt[j].m2;
else
    ts2 = ts1 + newprowt[j].m2;
    cs2 = 1;
}

    if (cs1 == 1 && cs2 == 1)
        temp = temp + ts2;

    score = temp;
}

return score;
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Vy kunta Lakshmi Kumar Adusumilli

Home Address:
969 E Flamingo Rd, Apt 167
Las Vegas, NV 89119

Degrees:
Bachelor of Technology, Electronics and Computers, 2003
Nagarjuna University, India

Thesis Title: Flow Shop Scheduling with Two Machines

Thesis Examination Committee:
Chairperson, Dr. Wolfgang Bein, Ph.D.
Committee Member, Dr. Lawerence Larmore, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Graduate Faculty Representative, Dr. Richard McCorkle, Ph.D.