INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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 bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI 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.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
On designing an AI based generic scheduling framework

Srinivasan, Venkatesh, Ph.D.
Case Western Reserve University, 1994
ON DESIGNING AN AI BASED GENERIC SCHEDULING FRAMEWORK

by

VENKATESH SRINIVASAN

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Advisor: Dr. Leon Sterling

Department of Computer Engineering and Science
Case Western Reserve University
May 1994
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

VENKATESH SRINIVASAN

candidate for the Ph.D.
degree.*

(signed)  

(chair)

date 2/18/94

*We also certify that written approval has been obtained for any proprietary material contained therein.
ON DESIGNING AN AI BASED GENERIC SCHEDULING FRAMEWORK

Abstract

by

VENKATESH SRINIVASAN

This thesis provides an answer to a question of practical significance: "Can a generic scheduler be designed that would significantly shorten the development time of individual schedulers?" In response this thesis presents a generic scheduling framework which makes it easier for scheduler developers to build customized scheduling applications, and provides end users a quick and easy means to test and implement new scheduling strategies.

The scheduling framework is organized as a collection of scheduler clusters where each scheduler cluster represents a class of schedulers solving related problems. Jobshop, transportation, timetabling, and process plant scheduler clusters are examples of scheduler clusters that are presented in great detail in this thesis, and specific schedulers namely carshop scheduler, tanker scheduler, examination scheduler, blending scheduler have been developed by instantiating the appropriate scheduler cluster following a systematic approach proposed in this thesis. The generic scheduler is essentially a generate and test system which resides behind an extensive user interface. During schedule generation, resources are allocated to tasks iteratively. Each iteration consists of choosing a task to be scheduled, choosing appropriate resources to be allocated, allocation of resources to tasks followed by constraint propagation which affects the remaining unscheduled tasks. This basic scheduling strategy is common to all schedulers developed within the framework.

Sample empirical results for three scheduler instances namely jobshop, transportation and timetabling are provided, these results show how the quality of the schedule generated depends on the problem instance, and choice of task and resource selection heuristics.
To R. Srinivasan and Late Rathna Srinivasan
my parents
for their love, affection and countless sacrifices
ACKNOWLEDGMENTS

It has been a privilege to work under the guidance of Leon Sterling who was my advisor and friend during my stay at Case Western Reserve University. It was his patience and encouragement which kept me going and finally enabled me to complete this thesis. Thanks also to Prof. Lee White, Prof. George Ernst and Prof. Vira Chankong for agreeing to be on my thesis committee. Dr. Bill Fabens of B.P. America, Warrensville, Ohio also had a great influence on this thesis and provided me with ideas to work on during the early stages of the thesis. In fact Leon and Bill Fabens wrote the first scheduler prototype and felt that this thesis was feasible.

I am thankful to the Department of Computer Engineering and Science, Case Western Reserve University for providing me a scholarship and supporting me as a T.A. during my first 3 years at the department. The Center for Automation and Intelligent Systems Research (CAISR) provided me with financial support for a year and also provided me computing resources during the course of my thesis. I also like to express my gratitude to B.P. America both at Warrensville, Ohio and Cleveland, Ohio. for providing me internships and jobs which added greatly to my experience and also helped me financially. Bill Fabens, Prabal Acharyya, David Osage were my benefactors and made working at B.P. America possible for me. I also very much enjoyed my discussions and company of my colleagues Ashish Jain, Ümit Yalçınalp, David Schwartz, Kathy Kovacic who kept the spirits of the PROSE group going.

During the course of my study I stayed at Steiner House which is an international CO-OP. I got a chance to meet many different people from about 15 different countries of the world. First I would like to thank Cornelia Rupp, who is a great person and who gave me constant support to survive the stress of finishing up the thesis and finally writing it down. There are too many people I met and made friends with during my stay at Steiner House, I can do justice to only a few. Upinder Singh, Yuko Hoshi, Mike Sabiers, Caner Yurteri, John Bontempo, Darcy Hamlin, Sunshine Squires, Uma Panchanathan, Mirmina Abraham, Fatih Uğurdag were some of my other good friends at Steiner House. Rahman Soomro, Shyam Sharan, Ashish Jain, Sanjay Srivastava, and Kumar Srinivasan were some other friends not living
in Steiner House who always provided me with good company.

Lastly I would like to thank my family members, my father, my brother Srinath and sister Chitra who put up with my long periods of silence and often wondered what I was doing in the U.S. and if I was ever going to finish my thesis.
Contents

1 Introduction ................................................................. 1
  1.1 Motivation .............................................................. 1
  1.2 Can Developing Schedulers be Systematized? ...................... 2
  1.3 Research Contributions .............................................. 3
  1.4 Generic Scheduling Framework Overview ............................ 4
  1.5 The Role of Prolog ...................................................... 7
  1.6 Organization of the Thesis .......................................... 9

2 Perspectives in Scheduling ............................................. 12
  2.1 Introduction ........................................................... 12
  2.2 The General Scheduling Problem .................................... 13
  2.3 Scheduling - an OR Perspective .................................... 14
       2.3.1 Schedule Measures ............................................ 15
       2.3.2 Schedule Parameters ......................................... 15
       2.3.3 Problems with Optimal Solutions ......................... 16
       2.3.4 Scheduler Classification .................................... 18
  2.4 Scheduling - a CSP Perspective ................................... 19
       2.4.1 Constraint Satisfaction Problem ............................ 19
       2.4.2 Approach to Solving CSP Problems ....................... 20
       2.4.3 Single Resource Scheduling .................................. 21
       2.4.4 Scheduling with Due Dates ................................ 24
       2.4.5 Activity with Precedence ................................... 24
       2.4.6 Multiple Alternative Resources ............................. 25
3 Design of a Scheduling Framework

3.1 A Generic Scheduling Framework

3.1.1 How to start developing a scheduler? 38

3.1.2 Scheduler Clusters 40

3.2 Specification of Scheduler Clusters

3.2.1 Task Specification 43

3.2.2 Resource Specification 45

3.2.3 Constraint Specification 46

3.2.4 Schedule Heuristics 47

3.2.5 Heuristic Execution 49

3.2.6 Schedule Parameters 50

3.2.7 Objective Specification and Costs 51

3.3 Schedule Generation 51

3.3.1 The Scheduling Engine 51

3.4 Schedule Generation 53

3.5 Handling Time in the Generic Framework 58

3.6 The Object System 62

3.7 Related Generic Software Environments

3.7.1 Software Abstraction 67

3.7.2 Framework for Domain Specific Software Design 68

3.8 Summary 71

4 Jobshop Schedulers 73

4.1 Introduction 73
4.2 Carshop Scheduling Problem ................................................... 75
4.3 Carshop Scheduling Components .............................................. 76
  4.3.1 Scheduler Input ................................................................. 77
  4.3.2 Scheduler Output ............................................................... 77
  4.3.3 Schedule Data ................................................................. 77
  4.3.4 Scheduler Constraints ....................................................... 79
  4.3.5 Scheduling Objectives ....................................................... 81
  4.3.6 Choice of Scheduling Measures ........................................... 82
4.4 Scheduler Components ........................................................... 82
  4.4.1 Task Dispatcher ............................................................... 82
  4.4.2 Resource Allocator ............................................................ 83
  4.4.3 Constraint Propagator ....................................................... 84
  4.4.4 Heuristics Selector ........................................................... 84
4.5 Grammar for Jobshop Schedulers .............................................. 84
  4.5.1 Grammar for Task Recognition ........................................... 86
  4.5.2 Grammar for Recognizing Resources ................................... 86
  4.5.3 Grammar for Recognizing Constraints ................................ 89
  4.5.4 Expressing Carshop Scheduler Heuristics ............................ 91
4.6 Scheduling Heuristics ............................................................ 92
  4.6.1 Heuristics for Task Selection ............................................ 92
  4.6.2 Operator Selection Heuristics ............................................ 93
4.7 Jobshop Scheduling Algorithm ............................................... 94
  4.7.1 Task Selection ............................................................... 96
  4.7.2 Choosing and Updating Operator Availability ....................... 101
  4.7.3 Constraint Propagation ................................................... 104
4.8 Empirical Results ............................................................... 104
4.9 Carshop Problem Traces ...................................................... 110
4.10 Summary ............................................................................ 124

5 Transportation Schedulers ...................................................... 125
  5.1 Introduction ........................................................................ 125

viii
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Land Based Vehicle Routing Problem</td>
<td>127</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Single Depot Vehicle Routing Problem</td>
<td>130</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Multi-Depot Vehicle Routing</td>
<td>132</td>
</tr>
<tr>
<td>5.3</td>
<td>Ocean Tanker Scheduling</td>
<td>134</td>
</tr>
<tr>
<td>5.4</td>
<td>A Tanker Scheduler Implementation</td>
<td>137</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Tasks: Specific Oil Delivery Requirements</td>
<td>137</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Resources</td>
<td>138</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Connectivity Matrix</td>
<td>138</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Constraints</td>
<td>139</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Costs</td>
<td>140</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Optimization Formulation for Tanker Scheduler</td>
<td>140</td>
</tr>
<tr>
<td>5.5</td>
<td>Grammar for Transportation Cluster</td>
<td>141</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Task Specification for a Transportation Scheduler</td>
<td>142</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Transportation Scheduler Resource Specification</td>
<td>144</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Transportation Scheduler Constraint Specification</td>
<td>146</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Transportation Scheduler Parameters Specification</td>
<td>147</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Extra Fields</td>
<td>149</td>
</tr>
<tr>
<td>5.6</td>
<td>Transportation Schedule Heuristics</td>
<td>150</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Pre-Defined Heuristics</td>
<td>150</td>
</tr>
<tr>
<td>5.6.2</td>
<td>User-Defined Heuristics</td>
<td>153</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Heuristic Execution</td>
<td>154</td>
</tr>
<tr>
<td>5.7</td>
<td>Generate and Test Algorithm</td>
<td>154</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Prolog Schedule Predicates</td>
<td>157</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Resource Selection:</td>
<td>157</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Allocation and updating:</td>
<td>159</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Routing</td>
<td>160</td>
</tr>
<tr>
<td>5.8</td>
<td>Results and Discussion</td>
<td>161</td>
</tr>
<tr>
<td>5.9</td>
<td>Conclusions</td>
<td>182</td>
</tr>
<tr>
<td>6</td>
<td>Timetabling Schedulers</td>
<td>185</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>185</td>
</tr>
</tbody>
</table>
6.2 Survey of Timetabling Applications .................................. 186
6.3 A Simple Timetabling Problem Formulation .................... 189
6.4 Vertex Coloring Problem ............................................ 191
6.5 Timetabling Cluster Design ......................................... 192
  6.5.1 Tasks ......................................................... 194
  6.5.2 Resources ................................................... 194
  6.5.3 Enrollment .................................................. 195
  6.5.4 Constraints .................................................. 196
6.6 Conflict List ...................................................... 196
6.7 Timetabling Schedule Generation .................................. 198
  6.7.1 Initialization ................................................ 198
  6.7.2 Task Selection .............................................. 199
  6.7.3 Resource Selection ........................................ 199
  6.7.4 Allocation .................................................... 202
  6.7.5 Constraint Propagation .................................... 202
  6.7.6 Calculations/Postprocessing ................................. 205
6.8 Timetabling Schedule Trace ....................................... 205
  6.8.1 Schedule Trace using Shortest Session First, Closest Fitting
       Resource First .............................................. 206
  6.8.2 Schedule Trace using Most Conflicting Task First, Earliest
       Schedulable Resource First .................................. 209
6.9 Session Scheduling Simulation Results ........................... 209
  6.9.1 Problem Descriptions ....................................... 212
  6.9.2 Scheduling using Shortest Task First ....................... 220
  6.9.3 Scheduling using Longest Task First ........................ 222
  6.9.4 Scheduling using Most Conflicting Task First ............. 223
  6.9.5 Scheduling using Least Schedulable Task First .......... 224
6.10 Conclusion ....................................................... 225

7 Process Schedulers .................................................. 233
  7.1 Introduction ................................................... 233
7.2 Problem Statement ........................................ 234
7.3 Blending Scheduler Details ............................... 235
  7.3.1 Tasks ................................................. 235
  7.3.2 Resources ............................................. 237
  7.3.3 Blending Recipes ..................................... 238
7.4 Constraints .................................................. 239
7.5 Process Scheduling Example .............................. 240
7.6 Process Scheduler Algorithm ............................. 243
7.7 Process Scheduler Heuristics ............................. 244
7.8 Conclusions ............................................... 245

8 Concluding Remarks ........................................ 247
  8.1 Contributions ........................................... 247
  8.2 Scheduler Clusters ....................................... 248
  8.3 On Designing Clusters ................................... 249
  8.4 Schedule Generation ...................................... 250
  8.5 Closing Remarks .......................................... 251
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Generic Scheduler Interface</td>
<td>5</td>
</tr>
<tr>
<td>1-2</td>
<td>Solution Enumeration Tree Structure</td>
<td>6</td>
</tr>
<tr>
<td>2-1</td>
<td>N-Castle CSP Problem</td>
<td>22</td>
</tr>
<tr>
<td>2-2</td>
<td>N-Castle CSP Problem (further variations)</td>
<td>23</td>
</tr>
<tr>
<td>2-3</td>
<td>Scheduling Steps in ISIS</td>
<td>28</td>
</tr>
<tr>
<td>2-4</td>
<td>OPIS-I Architecture</td>
<td>31</td>
</tr>
<tr>
<td>3-1</td>
<td>Generic Scheduler Interface</td>
<td>39</td>
</tr>
<tr>
<td>3-2</td>
<td>Scheduler Classification and Specific Scheduler Generation</td>
<td>41</td>
</tr>
<tr>
<td>3-3</td>
<td>Recognizing Schedule Input</td>
<td>43</td>
</tr>
<tr>
<td>3-4</td>
<td>DCG Recognizing Tasks</td>
<td>44</td>
</tr>
<tr>
<td>3-5</td>
<td>Sample Task Translation to Prolog Form</td>
<td>45</td>
</tr>
<tr>
<td>3-6</td>
<td>DCG Recognizing Resources</td>
<td>46</td>
</tr>
<tr>
<td>3-7</td>
<td>Sample DCG Recognizing Constraints For Scheduler Clusters</td>
<td>47</td>
</tr>
<tr>
<td>3-8</td>
<td>Example Heuristic Specification for a Transportation Scheduler</td>
<td>48</td>
</tr>
<tr>
<td>3-9</td>
<td>Grammar For Recognizing Heuristic Specification</td>
<td>50</td>
</tr>
<tr>
<td>3-10</td>
<td>Generic Scheduler Architecture</td>
<td>52</td>
</tr>
<tr>
<td>3-11</td>
<td>&quot;Generate and Test&quot; Scheduling Algorithm</td>
<td>55</td>
</tr>
<tr>
<td>3-12</td>
<td>Top Level Prolog Scheduling Predicate</td>
<td>56</td>
</tr>
<tr>
<td>3-13</td>
<td>Basic Prolog Scheduling Cycle</td>
<td>57</td>
</tr>
<tr>
<td>3-14</td>
<td>Example Time Representation of Events</td>
<td>59</td>
</tr>
<tr>
<td>3-15</td>
<td>Allen’s Temporal Relations</td>
<td>59</td>
</tr>
<tr>
<td>3-16</td>
<td>Some Prolog Temporal Predicates for Finding Intersecting Intervals</td>
<td>61</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3-17 Temporal Predicates in Prolog for Updating Intervals</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>3-18 Prolog Predicates for Accessing an Object System</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>3-19 Prolog Predicates for Accessing an Object System Continued</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>3-20 Overview of Software Design in KASE</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>4-1 Overview of Jobshop Scheduler</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>4-2 Generate and Test Algorithm for Jobshop Schedulers</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>4-3 Generate and Test Algorithm for Jobshop Schedulers Continued</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>4-4 Jobshop Schedule Generation</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>4-5 Dispatch Contention Window</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>4-6 Sample Prolog Predicates for Task Selection</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>4-7 Sample Prolog Predicates for Resource Selection/Update</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>4-8 Updating Resource Availability in Jobshop Schedulers</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>4-9 Sample Prolog Predicates for Constraint Propagation</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>4-10 Constraint Propagation in Jobshop Schedulers</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>5-1 Routing Clustered Proximal Tasks Together</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>5-2 Example Heuristic Specification for a Transportation Scheduler</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>5-3 Generate and Test Algorithm for a Transportation Scheduler</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>5-4 Generate and Test Algorithm for a Transportation Scheduler Continued</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>5-5 Sample Tanker Scheduling Predicates</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>5-6 Resource Allocation and Updating in Prolog</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>5-7 Prolog Predicates for Allocation</td>
<td>159</td>
<td></td>
</tr>
<tr>
<td>5-8 Sample Prolog Predicates for Route Generation</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>5-9 Plots a,b for Tanker Scheduler Dataset 1</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>5-10 Plots c,d for Tanker Scheduler Dataset 1</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>5-11 Plots a,b for Tanker Scheduler Dataset 2</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>5-12 Plots c,d for Tanker Scheduler Dataset 2</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>5-13 Plots a,b for Tanker Scheduler Dataset 3</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>5-14 Plots c,d for Tanker Scheduler Dataset 3</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>5-15 Plots a,b for Tanker Scheduler Dataset 4</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>5-16 Plots c,d for Tanker Scheduler Dataset 4</td>
<td>184</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

4.1 Sample Task/Resources in a carshop Scheduler .................. 78
4.2 Sample Operator Performance Data .............................. 78
4.3 Sample Resource Constraints for a carshop Scheduler .......... 80
4.4 Sample Carshop Task Translation ............................... 87
4.5 Sample Carshop Resource Translation ........................... 89
4.6 Sample Constraint Translation for Carshop scheduler ........... 91

5.1 Sample Task Requirement for a Tanker Scheduling Problem ...... 138
5.2 Sample Resources for a Tanker Scheduling Problem ............. 138
5.3 Sample Distance Matrix for a Tanker Scheduling Problem ...... 139
5.4 Task Field Description for Transportation Scheduler ........... 144
5.5 Resource Field Description for Transportation Scheduler ...... 146
5.6 Sample Constraints for a Transportation Scheduler ............. 148
5.7 Simplified Cost Information for a Transportation Scheduler ... 149

6.1 Enrollment Data for Dataset 1,2 .................................. 213
6.2 Tasks with All Fields - Dataset 1,2 .............................. 214
6.3 Conflict Matrix Generated for Dataset 1,2 ....................... 215
6.4 Conflict Matrix Generated for Dataset 1,2 Continued .......... 216
6.5 Resources Available for Schedule in Dataset 1,2 ................. 216
6.6 Conflict Matrix for Dataset 3 .................................... 217
6.7 Resources Available for Schedule in Dataset 3 .................. 218
6.8 Enrollment Data for Dataset 3 ................................... 218
6.9 Tasks with All Fields - Dataset 3 ................................ 219
7.1 Example Tasks for a Process Plant .................................. 235
7.2 Example Tank Specification in a Plant ................................. 237
7.3 Reaction Vessels in the Plant .......................................... 238
7.4 Sample Blend Recipes .................................................. 239

xvi
Chapter 1

Introduction

"Nothing in this world can take the place of persistence. Talent will not; nothing is more common than unsuccessful people with talent. Genius will not; unrewarded genius is almost a proverb. Education will not; the world is full of educated derelicts. Persistence and determination alone are omnipotent. The slogan "press on" has solved and always will solve the problems of the human race."

- Calvin Coolidge.

1.1 Motivation

Around July 1990 we were approached by a colleague working in R&D at a large corporation. He had been involved in writing several schedulers for clients within the corporation, covering a wide range of problems and each scheduler took 2-3 man years of development effort. Some of the problems he was involved in - were developing a production scheduler for manufacturing boron nitride slugs, wafers and powders; scheduling technicians to perform car repairs in a car shop, scheduling trucks dispatching oil to gas stations. While the task - scheduling, was similar in each case, very little software could be reused from one project to the next, and possibly adhoc strategies were used in developing each one of these schedulers. The language of development was also different in each case. The boron nitride slugs production
scheduler used the OPS5 language [SM90] (a rule based expert system) development language designed by Digital Equipment Corporation (DEC). For developing the truck dispatch scheduler Common Lisp and C were the two languages used in a MS-DOS environment while the jobshop scheduler was totally developed in Pascal on a PC under a MS-DOS environment. Each scheduler was developed by 2-3 people and when people left the organization, usage/extension of the schedulers reduced drastically due to a non standard design and different environments used in each of the schedulers.

1.2 Can Developing Schedulers be Systematized?

The question posed to us by the organization was: "Can a generic scheduler be designed that would significantly shorten the development time of individual schedulers?"

This thesis presents our response to the question - "a framework and methodology for scheduling". Historically we started with two ideas - one originally intended to develop a single program which we intended to call a "Generic Scheduler", the other idea was to develop a framework/environment where the development of the scheduler was semi-automatic and the developer had to customize code to develop individual scheduling applications.

The first approach needed input to the generic scheduler program in the form of a scheduler specification and the output of the generic scheduler was supposed to be a specific scheduler. On studying the problem more closely we discovered that such a generic scheduler or schedule generator was not really feasible due to the large variations in the scheduling problems, and invariably customizations needed to be made to specific scheduler instances. The second approach was then used to develop "A Generic Scheduling Framework/Environment" which allows a person to develop schedulers in a systematic manner with short turn around times. The development process is semi-automatic and requires the user to customize code for specific schedulers.
1.3 Research Contributions

During the development of the thesis we identified the common components of schedulers, and found that the diversity of concepts present in schedulers could better expressed by classifying schedulers into scheduler clusters grouped around common concepts. Schedulers from different clusters are developed with a common methodology. The environment we have developed has two parts, code common to all schedulers and code which varies between clusters.

The primary design goals set for and achieved by the end of this thesis are listed next:

- **Identify all the key entities common to a large group of practical and useful schedulers and build a representation scheme for expressing the key entities in these systems.** In response to this requirement the key entities in any schedule namely tasks, resources and constraints in the system, were identified and a generic framework to handle a group of practical schedulers is presented in Chapter 3.

- **Build a general purpose algorithm to handle scheduling problems.** In response to this objective a general purpose generate and test scheduling engine has been constructed in this thesis. This solves a wide variety of problems by generating solutions to the posed problems in an intelligent manner and tests the generated solutions for goodness of merit. The best solution is then taken.

- **Each scheduler should have the ability to generate alternative schedules.** Since the language of implementation was chosen to be Prolog - backtracking is an integral part of the system, where contrary to the typical manual and even automatic systems, it is possible to generate many solutions to choose from. The degree of backtracking can be specified and solutions both in the close neighborhood of the current solution as well as at other farther points in the solution space can be obtained after backtracking from the current schedule.

- **A unified programming strategy should be outlined to solve scheduling problems.** In response we have developed a top down problem solving approach.
We view the scheduling problem as "allocation of resources of tasks subject to constraints". Then scheduling is essentially to break the problem to tasks, resources and constraints and guiding the generate and test algorithm to allocate the identified resources to tasks.

- The thesis also provides a good method for developing prototype schedulers, and enables the end user to test the effect of different scheduling strategies on the performance of the system. He/She can build a strategy depending on the scheduling problem at hand.

- This thesis proposes a method for building programs for classes of schedulers and also a provides a mechanism to extend each cluster instance. Thus customization of schedulers is easier, for example classroom, examination, conference schedulers are basically customized instances of the same scheduler tailored to account for specific differences.

1.4 Generic Scheduling Framework Overview

Current software used for scheduling in industry is of limited flexibility and generality. This thesis presents a generic scheduling framework to alleviate the limitations. Our generic scheduling framework is designed to make it easy for scheduler developers to customize a scheduler for a particular application, and is flexible for end users generating a specific schedule. We take an Artificial Intelligence view of the problem of scheduling by concentrating on facilitating the representation of heuristics, rather than an Operations Research view concentrating on mathematical formulations. The scheduling framework is organized as a collection of scheduler clusters. Each scheduler cluster represents a class of schedulers solving related problems. Examples are job shop schedulers, transportation schedulers, and time tabling schedulers, each of these clusters have been presented in great detail in Chapters 4, 5 and 6 respectively.

Specific schedulers are developed by instantiating the appropriate scheduler cluster using an object-oriented interface, the design of a small object oriented system is presented in Chapter 3. Schedule generation proceeds in two phases. During the first phase, input data such as tasks to be scheduled, resources available and
heuristics for ordering tasks and resources are obtained from the user through a windowing interface and stored in an intermediate object form. During the second phase, a schedule is generated by a customizable scheduler written in Prolog. Our environment caters for two classes of people. Developers of schedulers can experiment with appropriate scheduling knowledge and end users can generate schedules. Overall, our approach is based on weak methods from artificial intelligence rather than mathematical programming methods from operations research. The scheduling
framework/environment envisaged is shown in Figure 1-1

Figure 1-2: Solution Enumeration Tree Structure

Scheduling is viewed as a typical search problem, where the solution is an optimized set of assignment of resources to tasks over time. The enumeration tree obtained while generating a schedule is shown in Figure 1-2. More details follow in Sections 3.1 and 3.1.2 in Chapter 3.

Chapter 3 describes our general approach to scheduling, the perspective is knowledge based allowing easy incorporation of heuristic knowledge guiding task dispatch and resource allocation. For the environment to be expandable and flexible, it must be straightforward to specify scheduler clusters. Our approach is to provide a
metalanguage and such a language is implemented in the system by writing DCG grammar rules.

Each scheduler is built around an iterative procedure. One iteration consists of choosing a task, choosing a resource, checking constraints, and propagating the effects of the allocation of the resource to the task for constructing the rest of the schedule.

1.5 The Role of Prolog

The code presented in this thesis is implemented in Prolog. Prolog was the language of choice for developing the generic scheduling framework for a variety of reasons. In this thesis stubs of Prolog codes have been presented in various chapters. The main idea of presenting Prolog code is to illustrate to the reader the top down programming methodology used in this thesis, and also provide an idea how Prolog is used to develop scheduling code. The user can skip portions of Prolog code is he/she wishes to do so, and still understand the algorithm which is presented in a Pascal style pseudo code in each chapter. Some of the reasons for choosing Prolog as the development language are enumerated next:

- Prolog is not a strongly typed language. This is useful as the range of problem whose solution is attempted here is large. Prolog is an easily extensible language and in the current implementation an object system has been built on top of Prolog. More details of the object system are presented in Section 3.6. Development time would be considerably larger if a conventional language like 'C' was used for this purpose.

- The scheduling algorithm used is a generate and test scheduling algorithm. Generate and test is a common technique in algorithm design and programming. In generate and test one process, or routine, generates a set of candidate solutions to the problem, and another process, or routine, tests the candidates, trying to find one, or all, of the candidates which actually solve the problem. It is easy to write logic programs that, under the execution model of Prolog, implement the generate and test technique. Such programs typically have a
conjunction of two goals, in which one acts as the generator and other tests whether the solution is acceptable, as in the following clause:

\[
schedule(S) \leftarrow generate\_schedule(S), test(S).
\]

A standard technique for optimizing generate and test programs is to try and "push" the tester inside the generator as "deep" as possible. This has also been done in this thesis where constraints have been used to prune the search space.

- Another very important aspect for the generic scheduling framework is the ease of data representation for a variety of schedulers. Prolog is very well suited for this purpose and in fact it originated from attempts to use logic to express grammar rules and formalize parsing process. The most popular approach to parsing in Prolog is definite clause grammar or DCGs. DCGs are a generalization of context free grammars that are executable, because they are a notational variation of a class of Prolog programs. Context free grammars consist of a set of rules of the form

\[
<\text{nonterminal}> \rightarrow <\text{body}>.
\]

where nonterminal is a nonterminal symbol and the body is a sequence of one or more items separated by commas. Each item is either a nonterminal symbol or a sequence of terminal symbols. Grammar for recognizing tasks, resources and constraints for scheduler clusters has been presented in this thesis.

Prolog can be used to build good scheduling applications and some other known efforts to develop scheduling systems based on Prolog Based Tools have been made in the framework of the Eureka Project PROTOS [Pro88]. Recently other interesting scheduling applications written in Prolog have been reported, these include a scheduling system for an aircraft fleet [M+92], and another one for scheduling of Ship Hull production [G+92].

Now a brief overview of the organization of this thesis is provided.
1.6 Organization of the Thesis

The thesis is organized into eight chapters. The current chapter which is the first chapter presents a brief overview of the thesis, a list of contributions made by the current research and the organization of the thesis.

Chapter 2 titled "perspectives in scheduling" presents two perspectives on scheduling - an OR perspective and "scheduling modeled as a constraint satisfaction problem". Constraint satisfaction problems have been actively pursued by the artificial intelligence community. The requirement here is to find a set of values for variables each of which belong to particular domains and are constrained with respect to each other. Nomenclature used to describe the problems in both domains is presented, along with some of the problems whose solutions are known to be obtainable in polynomial time.

The second chapter also contains a discussion on a group of schedulers developed at Carnegie Mellon University. It also contains a brief discussion on some non jobshop schedulers and scheduling methods, and ends with a discussion on the representation of time in the generic framework.

Chapter 3 presents our answer to the question "Can a generic scheduler be designed that would significantly shorten the development time of individual schedulers?". The design of a generic scheduling framework is presented whose purpose is to make it easy for scheduler developers to customize a scheduler for a particular application, and also make it flexible for end users to generate specific schedules easily. Here we have identified the common components of schedulers and found that the diversity of concepts present in schedulers could be better expressed by classifying schedulers into scheduler clusters, grouped around common concepts.Schedulers from different clusters are developed with a common methodology. Each scheduler cluster represents a class of schedulers solving related problems.

One of the main aims of this thesis is to synthesize software for developing a range of schedulers. We present a discussion on other related generic software environments and discuss the KASE (Knowledge Assisted Software Engineering) project. The goals of KASE and this dissertation are very similar i.e. to provide software abstraction and develop software for scheduling.
The following chapters are specific schedulers developed to illustrate the concepts presented in Chapter 3.

In Chapter 4 the formulation of a cluster which we call the "Jobshop scheduling cluster" is described. One instance of the jobshop cluster called the carshop scheduling problem has been presented in great detail here. The ideas and algorithm used in the jobshop scheduling cluster provides a method to handle problems in the same class, as well as methods which can be re-used for other scheduling clusters developed later on in this thesis. The jobshop scheduling cluster solves a very common kind of scheduling problem - there are \( n \) jobs \( \{J_1, J_2, \ldots, J_n\} \) which have to be processed/scheduled using \( m \) resources \( \{R_1, R_2, \ldots, R_m\} \). The resources in the case of a carshop scheduler are repair operators while the jobs are car repair jobs.

Chapter 5 presents the design of the "Transportation scheduling cluster". The essential notion of transportation scheduling is to efficiently allocate transportation assets or resources (e.g. planes, ships, trucks) to some movements (e.g. bulk cargo, passengers) over time intervals in a manner so as to minimize the total transportation cost which is an indicator of the "goodness" of the schedule. All allocations are subject to constraints which dictate if specific assignments of transporters to movements are feasible or not. In chapter 5 we discuss two primary kinds of transportation problem which have attracted attention, the first one being land based vehicle routing problems, and the other one being ocean tanker scheduling.

An ILP (integer linear program) formulation for the land based vehicle routing problem and ocean transport scheduling problem is presented. From the formulations one can see how even the simplest of problems gives rise to a large number of variables and it is computationally very hard if not infeasible to solve the problems. This chapter contains our solution to the transportation problem which specializes the generic generate and test scheduling algorithm proposed in Chapter 3 to the transportation cluster. Following the philosophy of the research; tasks, resources and constraints are identified for the current problem. Each one of the tasks, and resources are represented as objects and described in greater detail in this chapter.

Chapter 6 contains the description of a scheduling cluster called the "Timetabling scheduling cluster". The timetabling problem can be defined as scheduling of a cer-
tain number of sessions which must be attended by a specific groups of enrollees, over a definite period of time, requiring certain resources. Some common examples of timetabling problems are classroom scheduling, examination scheduling and conference scheduling. In classroom scheduling the sessions are classes which have to be attended by students who are the enrollees. The resources are teachers, rooms and teaching aids for the classroom to be held.

The timetabling schedulers developed in this thesis are based on the framework presented in Chapter 3. Scheduling here is viewed as allocation of venues to sessions, where venues and sessions have been identified as resources and tasks, and a session constitutes a common group of enrolled people. A formal problem formulation for the timetabling schedulers is provided later, it suffices to say here that the timetabling scheduling cluster mainly tries to minimize conflicts during the schedule. Conflicts in scheduling has been treated as defining an undirected linear graph independent of the relation of the activities; in conflict to additional constraints of time and space. Each connected component of such a graph corresponds to a set of events that must be scheduled at different times.

Chapter 7 named “Process Schedulers” contains the description of a scheduling cluster called the process scheduling cluster, Process plant scheduling is not reported widely in the OR literature, and the basic ideas expressed in this chapter were in response to solving a real life problem which arose in the petroleum industry. The methods developed here can be used to model any process plant scheduling problem, which is essentially a flow problem consisting of solid, liquid or gas flow, unit operations like heat transfer, mass transfer and finally the flow of the final product out.

Chapter 8 named “Concluding Remarks” presents a short summary of the thesis including the problems solved and contributions which would make this thesis useful for people developing schedulers.
Chapter 2

Perspectives in Scheduling

A comment on schedules:

Ok, how long will it take?

For each manager involved in initial meetings add one month.
For each manager who says "data flow analysis" add another month.
For each unique end-user type add one month.
For each unknown software package to be employed add two months.
For each unknown hardware device add two months.
For each 100 miles between developer and installation add one month.
For each type of communication channel add one month.
If an IBM mainframe shop is involved and you are working on a non-IBM system add 6 months.
If an IBM mainframe shop is involved and you are working on an IBM system add 9 months.
Round up to the nearest half-year.

—Brad Sherman.

2.1 Introduction

Scheduling has been a major research topic for many years. It started out primarily as a topic in operations research, and since AI research is concerned with general reasoning processes, it can and has contributed greatly to the advancement
of scheduling technology. Scheduling is a very broad and diverse topic now, making it difficult to review the entire literature of scheduling in depth. This chapter has been named "perspectives in scheduling" where scheduling is introduced from both the operations research and constraint satisfaction problem point of view. This chapter is organized as follows, First a general discussion is presented on scheduling and on different domains in which we come across scheduling. Next two perspectives on scheduling are presented - an OR perspective and the AI perspective modeling "scheduling as a constraint satisfaction problem". Constraint satisfaction problems are described in more detail in Section 2.4 and have been actively pursued by the artificial intelligence community. In this section nomenclature used to describe the problems in both OR and constraint satisfaction problem (CSP) domain has been presented, along with some of the problems whose solutions are known to be obtainable in polynomial time.

This is followed by a discussion of a group of schedulers developed at Carnegie Mellon University. Why CMU Schedulers? - the answer to the question lies in the fact that these schedulers are well documented and have constantly evolved in the mid eighties to the present. We also present a brief discussion on some other schedulers and scheduling methods which have been developed and have contributed to the field of scheduling. The current chapter is like a review chapter which presents diverse ideas on scheduling, while in Chapter 3 a generic scheduler architecture is presented which is followed by scheduler instantiations in Chapters 4, 5, 6, and 7.

2.2 The General Scheduling Problem
Scheduling deals with the allocation of resources over time to perform a collection of tasks [Bak74], [Fre82a], [Kan76]. Scheduling problems arise in many domains. In the manufacturing domain, tasks are referred to as jobs, correspond to parts or batches of parts that need to be processed on a set of machines [MT63], [JM74], [Gra81], [SP85]. In hospitals tasks are patients and resources are nurses, hospital beds or medical equipment required to treat the patients. Scheduling problems arise in schools, where the tasks are classes and resources can be teachers, classrooms and students [FG89], [DR86]. Other examples include transportation related problems
(e.g. troop transportation, airport terminal scheduling), computer scheduling problems (e.g. CPU scheduling [PS74], space telescope scheduling [Joh90]). Scheduling problems are formulated as either Constraint Satisfaction Problems (CSPs) or Constrained Optimization Problems (COPs). A COP is a CSP with an objective function to be optimized subject to problem constraints [PS82], [NW88], [FSB89]. A general paradigm for solving CSPs and COPs relies on the use of backtrack search [GB65], [BR75]. Within this paradigm, the scheduling problem is solved through selection of a subproblem and the tentative assignment of a solution is reached that cannot be completed without violating a problem constraint, or one or several earlier assignments need to be undone. This process of undoing earlier assignments is called backtracking.

Traditionally, scheduling techniques have dealt with backtracking by transforming the mathematical structure of the problem, and allowing some constraints to be relaxed as needed. This approach is commonly used in factory scheduling, where rather than requiring that all jobs be completed by their due dates, job due dates are relaxed as much as necessary in order to efficiently come up with a schedule. While producing efficient scheduling procedures, this approach often results in fairly poor schedules, because a large number of constraints are violated. Before we go on to develop a set of schedulers we present a brief description of the work which has been done in the area of jobshop scheduling in operations research, followed by some recent work done in the area of constraint satisfaction problems which is useful towards designing and building new schedulers.

2.3 Scheduling - an OR Perspective

Operations Research has focussed on understanding the variety of scheduling environments that exist, and constructing environment specific algorithms. In this section we refer to some common scheduling problems which have been well studied in OR. We do not make use of all these results in this thesis; rather we use some of the concepts developed. In the OR literature, there are four types of "shops", which are distinguished based on job operations:
• single machine - single operation.

• parallel machine - single operation.

• flow shop series of machines - multiple operations.

• job shop network of machines - multiple operations.

A job is defined as having

• One or more operations.

• A processing time for each operation.

• A due date.

2.3.1 Schedule Measures

The utility of the schedule is measured in terms of three major objectives: Minimize work in progress inventory, maximize utilization of personnel/facilities and maximize customer service. Specific objective functions to minimize may be:

Mean flowtime of completion time: each job \( j \) has a completion time \( c_j \) and \( \sum_{i=1}^{n} c_j/n \) is the average completion time.

Tardiness: the tardiness of job \( j \) is \( d_j - c_j \) where \( d_j \) is the due date of job \( j \) and \( c_j \) is the completion time of job \( j \).

Lateness: the lateness of job \( j \) is \( |(d_j - c_j)| \) where \( d_j \) is the due date of job \( j \) and \( c_j \) is the completion time of job \( j \).

Makespan: the makespan of a schedule is \( c_j - s_i \) where \( c_j \) is the completion time of last completing job \( j \) and \( s_i \) is the start time of the first job \( i \).

2.3.2 Schedule Parameters

Four parameters \( n/m/A/B \) have to be specified in order to describe a jobshop problem
is the number of jobs.

m is the number of machines.

A describes the flow pattern or discipline within the machine shop. When m=1, A is left blank.

F For the flow-shop case, i.e., machine order for all jobs is the same.

P For the permutation flowshop case. Here not only is the machine order the same for all jobs, but now we restrict the search to schedules in which the job order is the same for each machine.

G The general jobshop case where there are no restrictions on the form of constraints.

B describes the performance measure by which the schedule is to be evaluated. Makespan, tardiness, cost are some common scheduling measures.

### 2.3.3 Problems with Optimal Solutions

Most problems with optimal solutions are simple problems. The optimal results typically are valid for one machine, or at most 2-3 machines. Moreover even for 3 machine problems the optimal results are obtainable only in very specific cases. Some known results for single machine problems are stated in brief in this section and the interested reader should see [Bak74] and [Fre82a] for detailed results. The main purpose of stating some of the known results is to give a feel of OR solutions to solvable problems.

\[
\begin{align*}
    p_j &= \text{Processing time for task } j \\
    r_j &= \text{Arrival, ready or release time for task } j \\
    d_j &= \text{due date or deadline of } j \\
    w_j &= \text{cost per unit time of delay, weight, importance}
\end{align*}
\]
n = # of tasks (static problem)
Wj = Waiting time or delay preceding j
Cj = completion time of j = rj + pj + Wj
Lj = Lateness (cj - dj)
Tj = Tardiness = max(0, cj - dj)
Fj = Flow time = pj + Wj

Property 1: (SPT Schedule) For 1 | - | Σcj, ΣLj, or ΣFj i.e. single machine problem with the objective of minimizing total completion time, or lateness or total flowtime, scheduling the shortest processing time jobs first yields optimal solutions.

Property 2: For 1 | - | ΣTj, SPT (shortest processing time) is optimal if, when using it, Cj+1 ≥ dj + pj holds good for each job. This means that it is enough that all jobs be tardy if we give each job, the next higher processing time.

Property 3: For 1 | dj = d | ΣTj, SPT is optimal.

Property 4: (Smith’s ratio rule) For 1 | - | ΣwjCj arranging tasks such that pj/wj is increasing (shortest weighted processing time); is optimal.

Property 5: For 1 | (pmnt) | max(wjCj), arranging tasks with decreasing wj is optimal. This means that for minimizing the weighted completion times, the processing times are irrelevant and scheduling jobs with the job having maximum weight first is optimal.

Property 6: (Jackson’s Rule) For 1 | - | Lmax or Tmax, arranging tasks with increasing due dates dj is optimal. Again for this objective, the processing time of each job is not important.

Lawler’s Theorem: For 1 | (pmnt) | fmax, schedule job k last where
fk(p) = minj fj(p) (p = Σjpj)
That is, process last the job that is least costly in that position.

Other important results for single processor problems include the case when precedence constraints come in to play. One important algorithm developed for handling precedence constraints is called the “parallel chain algorithm”. The basic
idea of the parallel chain algorithm used here is to change a problem into an equivalent preference ordering problem. Task $i$ has preference over task $j$ if, whenever $i$ and $j$ appear together in any schedule, we are at least as well off sequencing $i$ before $j$ if it is feasible to do so. For the Flowshop scheduling problem the most celebrated rule is Johnson's rule, this rule is applicable to a $2F | - | C_{\text{max}}$ type problem.

(Johnson's Rule): This rule applies to a two processor flow shop problem.

$$A_j = \text{Processing time of job } j \text{ on processor 1.}$$
$$B_j = \text{Processing time of job } j \text{ on processor 2.}$$

Let $\mathcal{F} = \{ j : A_j < B_j \}$, and $\mathcal{L} = \{ j : A_j \geq B_j \}$, then schedule the jobs in $\mathcal{F}$ first in a non-decreasing order of $A_j$, followed by the jobs in $\mathcal{L}$ in non-decreasing order of $B_j$. Addition or removal of jobs does not affect the optimal ordering of the rest.

Other problems especially involving more than 3 machines, mostly do not have any optimal solutions. The common techniques used for solving the problems are using branch and bound and dynamic programming techniques. As the size of the problem increases, variables in the problems increase and these mathematical techniques become unwieldy.

It is important to note that such simple results do not exist in the domain of transportation, timetabling, process scheduling domain. One of the reasons is that a single vehicle problem, or a timetabling a single session are not of any practical interest. In these domains most of the results have been obtained through simulation and most problems are NP-Complete.

### 2.3.4 Scheduler Classification

Classifying scheduling problems has been attempted in the OR literature. For instance, Lageweg, Lawler, Lenstra and Rinooy Kan [LLK82] have categorized a class of deterministic machine scheduling problems, Bodin and Golden [BG81], [DLS90] have outlined a classification scheme for vehicle routing and scheduling problems.
Several good papers exist which survey the area of time tabling problems, notable amongst them is one by Michael Carter [Car86] which surveys practical applications of examination timetabling. Carter has actually classified classroom scheduling problems and presented different cases when scheduling is easy and when scheduling is difficult. This thesis also classifies schedulers according to different domains and such a classification scheme is presented in Section 3.1.2.

2.4 Scheduling - a CSP Perspective

In this section we view the scheduling problem as a constraint satisfaction problem. First a constraint satisfaction problem is defined, after which the common methods for solving constraint satisfaction problem are presented. Next we present a job-shop scheduling problem modeled as a constraint satisfaction problem, such that the solution to the constraint satisfaction problem is the solution to the jobshop scheduling problem.

2.4.1 Constraint Satisfaction Problem

A constraint satisfaction problem is defined by a set of variables \( V = \{v_1, v_2, \ldots, v_m\} \) each having a corresponding domain \( D = \{d_1, d_2, \ldots, d_m\} \), and a set of constraints \( C = \{c_1, c_2, \ldots, c_n\} \). A constraint \( c_j \) is an \( m \)-tuple that specifies a consistent assignment to the variables i.e., \( c_j \subseteq d_1 \times d_2 \times \ldots \times d_m \). A variable's domain \( d_i \) can be infinite, but is usually discrete and small. CSP takes a reductionist approach to problem solving [Sim83]; in the reductionist approach a superset of solutions are successively reduced to a solution set. The process of solving a CSP is comprised of the following steps:

1. Select a variable for instantiation.

2. Select a value to assign to the variables, and then

3. Determine whether the assignment is consistent with all the constraints. If not, then backtrack, otherwise iterate.
The general CSP is NP-complete [GJ79]. Techniques for solving the general CSP extend the depth-first backtrack search procedure in which a solution is incrementally built by instantiating one variable (or more generally one subproblem) after another [GB65, Pea63]. Every time a variable is instantiated a new search state is created, where new constraints are added to account for the value assigned to that variable. If a partial solution is built that cannot be completed, the current search state is said to be a deadend. The system then needs to backtrack to an earlier less complete solution, and try alternative variable assignments. Search typically stops when a first solution has been found, or when all the alternatives have been tried without any success. In the latter case the CSP is said to be infeasible.

2.4.2 Approach to Solving CSP Problems

Because the general CSP is NP-complete, backtrack search may require exponential time in the worst case. Research in CSP has produced three types of techniques that can help improve the average efficiency of the basic backtrack procedure [DP88].

Research has focussed on heuristics for selecting variables and values, some of the common heuristics cited in the literature are briefly mentioned here. We have based our discussion on jobshop scheduling problems, the same discussion is valid for other scheduling problems as well, as will be seen in Chapters 3, 4, 5 and 6.

1. Consistency Enforcing (Checking) Techniques: These techniques are meant to prune the search space by eliminating local inconsistencies that cannot participate in a global solution [Mac77]. This is done by inferring new constraints and adding them to the current problem formulation. If during this process the domain of a variable becomes empty, a deadend situation has been identified. There is however a tradeoff between the amount of computation spent enforcing consistency and the savings achieved in the actual search [Mac77], [Nad88]. Partial consistency enforcing algorithms have been classified according to the degree of consistency; in particular consistency enforcing algorithms that achieve consistency among a set of $k$ variables are said to enforce $k$-consistency [Fre82b].
2. Variable and Value Ordering Heuristics: These heuristics are concerned with the order in which variables are instantiated and values assigned to each variable. A good variable ordering is one that starts with variables which are hardest to instantiate, thus hopefully the partial solutions constructed do not lead to deadends. A value ordering heuristic is one that leaves as many options as possible to the remaining uninstantiated variables. These heuristics are meant to reduce the cost of backtracking and its cost when it cannot be avoided. Both variable and value ordering heuristics can significantly reduce search [HE80], [Pur83], [DP88].

3. Deadend Recovery Techniques: These techniques help decide which earlier assignments to undo in order to recover from a deadend. The simplest such strategy is known as chronological backtracking. It consists of undoing the last assignment, and trying another one if one is left. More sophisticated deadend recovery strategies have been designed that attempted to go back to the source of failure and undo one or several of the assignments that prevent the current partial solution from being successfully completed [SS77], [Doy79], [Dec89]. Techniques have also been developed that attempt to "learn" from deadends by abstracting from these situations a set of partial assignments that are inconsistent and should be avoided in the future [Dec89].

The CSP paradigm scales up pretty well to the jobshop and other scheduling problems. Improved heuristics to be used under the CSP paradigm have been proposed in Norman Sadeh's thesis [Sad91].

2.4.3 Single Resource Scheduling

The simplest factory that one could imagine scheduling contains a single machine and produces a single product that requires a single operation. The scheduling goal is to assign each order for a product to an available time slot on the machine. Examples of the map between a factory scheduling problem and a constraint satisfaction problem are presented in a paper by Mark Fox and Norman Sadeh [FS90].
Equivalently, this can be stated more generally as a resource allocation problem where a single, indivisible resource, is to be allocated over time to n activities, but at any time to at most one activity. The activities are unrelated (i.e., no precedence relations between them) and are of equal duration.

![Diagram](image)

(a) N-Castle Problem  
(b) Mutated N-Castle Problem

Figure 2-1: N-Castle CSP Problem

From a CSP perspective, this is equivalent to a N-Castle Problem. Given an n x n chess board, the problem is to place n castles such that they do not interfere according to chess rules. Unlike the N-Queens problem more than one castle can occupy the same diagonal, but not the same row or column. Each castle, which occupies a separate row, corresponds to a separate activity and each column of the chess board corresponds to a unique, equal duration time slot that the activity can use as a single resource. The representation of the N-castle problem modeling a single resource scheduling problem in shown in Figure 2-1(a).
More formally, given $n$ activities $A_i$ with domains $A_i \in \{1, \ldots, n\}$, assign a value to each, subject to the following constraints:

- $\forall \ ij \ [(i \neq j) \Rightarrow (A_i \neq A_j)]$

No two distinct activities may occupy the same column.

The constraint graph, whose nodes are activities, is completely connected by the inequality constraints that assure that no activities can occupy the same column/time slot. This problem is simpler than the N-Queens problem and can be solved in polynomial time [GJ79].
2.4.4 Scheduling with Due Dates

A more realistic problem is to impose constraint that each activity $A_i$ must be completed before due date $d_i$. Each activity's due date is independent of the due date of other activities. This is the same as "mutilating" the N-Castle chess board by removing squares at the end of each row. Assuming that each column is numbered from 1 to n, then if an activity $A_i$ is due on date 5, then squares 6 through n in the activities corresponding row are unavailable for placing the castle. The mutilated N-Castle problem is shown in Figure 2-1(b).

More formally, given n activities $A_i$ with domains $A_i \in \{1, \ldots, n\}$, and due dated $d_i$. Assign a value to each subject to the following constraints

- $\forall \ ij \ [i \neq j \supset (A_i \neq A_j)]$
  
  No two distinct activities may occupy the same column.

- $\forall \ i \ [A_i \leq d_i ]$
  
  An activity must end on or before its due date.

2.4.5 Activity with Precedence

Consider the factory where each product is produced according to a process plan. A process plan defines a sequence of operations (or activities) that must be performed in the order specified.

More generally, the single resource scheduling problem is now further complicated by imposing precedence among activities. The mutilated N-castle problem is shown in Figure 2-2(a) and is formally defined as given n activities $A_i$:

- with domain $A_i \in \{1, \ldots, n\}$,
- due dates $d_i \in \{1, \ldots, n\}$, and
- a precedence matrix $P_{ij}$ where $P_{ij} = 1$ if $A_i$ must precede $A_j$.

Assign a value to each subject to the following constraints:
- $\forall ij \ [(i \neq j) \supset (A_i \neq A_j)]$
  
  No two distinct activities may occupy the
  same column.

- $\forall i \ [ A_i \leq d_i ]$
  
  An activity must end on or before its due
date.

- $\forall ij \ [(P_{ij} = 1) \supset (A_i < A_j)]$
  
  Activities with precedence must be se-
  quenced.

Initially, the constraint graph had all of the activity nodes connected together
with inequalities, denoting that no castle/activity may occupy the same position.
Precedence adds another layer of inter-activity constraints, as denoted by the $P_{ij}$
precedence matrix.

2.4.6 Multiple Alternative Resources

To make the factory a little more realistic, more resources can be added. Now, each
activity $A_i$ may choose one of the $m$ resources to use, thus increasing the complexity
of the task.

From the CSP perspective, the N-Castle problem can be further refined by ex-
tending the $n \times n$ into a third dimension, each plane representing an alternative
resource as shown in Figure 2-2(b). No two castles may occupy the same column
within the same plane.

More formally, given $n$ activities $A_i$ and $m$ alternative resources to choose from,
with:

- activities having domains $A_i \in \{< T_i, R_i >| 1 \leq T_i \leq n, 1 \leq R_i \leq m\}$ where
  $T_i$ is the time of column position of the activity and $R_i$ is the resource or plane
  selected,

- due dates $d_i \in \{1, \ldots, n\}$, and

- a precedence matrix $P_{ij}$ where $P_{ij} = 1$ if $A_i$ precedes $A_j$, 
assign a value to each subject to the following constraints:

- due date and precedence constraints as in
  the previous problem.

- \( \forall \ ij \ [((i \neq j) \land (R_i = R_j)) \supset (T_i \neq T_j)] \).

No two distinct activities using the same re-
source may occupy the same time slot.

Nodes in the constraint graph are now 2-tuples, significantly enlarging the space
of alternative potential solutions.

In the previous subsections a brief translation between scheduling and constraint
satisfaction problems has been provided. These show how ideas used in solving
constraint satisfaction can be used to solve the scheduling problems as well.

2.5 Previous Work

One of the notable efforts in building scheduling systems has been undertaken at
Carnegie Melon University, which started in the late seventies. Scheduling had
been mostly dealt with from an OR perspective where a collection of problems were
known to have optimal solutions, and most other problems were solved by either
using approximations or using dispatch rules (a dispatch rule is a local decision
which determines the next job to be processed on a machine from the set queued
on the machine). The first attempt at CMU was made by Mark Fox whose thesis
"Constraint-Directed Search: A Case Study for Jobshop Scheduling" [Fox87] investi-
gates the problem of constraint-directed reasoning in the jobshop scheduling domain.

By viewing the scheduling process from a constraint directed search perspective,
much of the knowledge can be viewed as constraints on schedule generation and
process selection. Other issues discussed are issues on knowledge representation se-
manics for organizational modeling, extending knowledge representation techniques
to include the variety of constraints found in the scheduling domain, integrating the
constraints into the search process, and relaxing constraints when conflicts occur.
The ISIS project is described in greater detail in this section. The ISIS-I and ISIS-II
project culminated in 1983, followed by OPT [Jac84], after which the next important project based on opportunistic constraint directed search were OPIS-I and OPIS-II [OS88] which introduce multiple scheduling perspectives based on both order and resource. The performance of OPIS schedulers was better than ISIS schedulers. OPIS was succeeded by CORTES [SF89] and MICRO-BOSS [Sad91], both of which try to increase the opportunism in scheduling from a macro-opportunistic level to a micro-opportunistic level.

## 2.5.1 ISIS

The ISIS factory scheduling system developed by Mark Fox and his team first demonstrated the potential of AI modeling and heuristic search techniques to help solve production scheduling problems [Fox87], [SFO86]. For the first time, rather than relying on a simplified model of the shop, ISIS attempted to deal with the full range of constraints and objectives encountered in the manufacturing domain and implements a constraint-directed search approach in its construction of jobshop schedules. Though this approach is complex it can be basically viewed as a variation of generate and test. That is, there exist a set of operators which define a space of states which represent partial and full searches. The view taken in ISIS concerning the representation and utilization of this domain knowledge is that it can be represented in a single formalism of constraints. That is knowledge such as:

- organizational goals,
- physical Characteristics and capabilities,
- causality,
- preference of action and selection, and
- availability of resources,

may all be viewed as constraints. Additionally, there may exist knowledge at the search level which may constrain the reasoning process. For example, certain types of orders, e.g., forced outage, may be scheduled in a manner differently than
other orders, e.g., shop orders. These can be viewed as meta constraints since they operate at a level above the scheduling problem.

The goal of ISIS is to construct schedules which satisfy as many constraints as possible in near realtime. To achieve this, ISIS uses constraints to bound, guide and analyze the scheduling/search process. The ISIS system performs a hierarchical, constraint directed search in the space of alternative schedules. There are four levels in the system.

![Diagram of scheduling steps in ISIS](image)

Figure 2-3: Scheduling Steps in ISIS
Lot Selection (Level 1): Selects an order to be scheduled according to the prioritization algorithm based on the category of the order and its due date.

Capacity Analysis (Level 2): Performs the capacity analysis of the plant. It determines the earliest start time and the latest finish time for each operation of the selected order, as bounded by the order's start and due date. The times generated at this level are codified as constraints which are passed to level 3.

Resource Analysis (Level 3): In this level a detailed scheduling of all resources necessary is performed to produce the selected order. Pre-search analysis begins by examining the constraints associated with the order to determine the scheduling direction (forward vs backward), whether any additional constraints should be created (e.g., due dates, work in progress) and the search operators which will generate the search space. A beam search is then performed using the selected search operators. Each application of an operator generates another "ply" in the search space. At any ply only the "n" highest rated states are selected for extension to the next ply.

Reservation Selection (Level 4): The fourth level takes as input time bounds on each of the resources required by the operation in the operation sequence selected at level three. It selects a time reservation for each of the resources which attempts to minimize the work-in-progress time of the lot. Figure 2-3 shows the four stages during scheduling in the ISIS-1 scheduler. Search takes place in all the levels and results from a lower level are used in the next higher level in the form of bounding constraints.

The search procedure requires jobs to be scheduled one by one (so called job centered approach). While this search procedure was particularly efficient in reducing inventory, it had problems optimizing utilization of bottleneck resources. In order to improve on the problem the OPIS schedulers were developed.

2.5.2 OPIS

OPIS was developed by Steve Smith and Peng Si Ow [SFO86], [OS88]. In OPIS, the notion of bottleneck resource was pushed one step further as it was recognized
that new bottlenecks can appear anywhere during the construction of the schedule. The OPIS scheduler combines the two scheduling perspectives: a resource-centered perspective for scheduling bottleneck resources, and a job-centered perspective to schedule non-bottleneck operations on a job by job basis. Rather than relying on its initial bottleneck analysis, OPIS typically repeats this analysis each time a resource or a job has been scheduled. The ability to detect the emergence of new bottlenecks during the construction of the schedule and revise the current scheduling strategy has been termed as opportunistic scheduling [OS88]. An overview of OPIS-I is presented in Figure 2-4.

The OPIS design differentiates between intra-order and inter-order interactions. The intra-order interactions refer to the interactions induced by the activity precedence constraints, whereas the inter-order interactions are the ones induced by the resource capacity constraints. Order scheduler (OSC) and Resource Scheduler (RSC) units are shown in Figure 2-4; current schedules are maintained by the Schedule Maintenance Subsystem. The opportunism lies in the ability of OPIS to shift from one problem solving perspective to another one as it builds the schedule. At each stage of the schedule analysis is performed by the Conflict Analyzer and Capacity Analyzer. The Search Manager is guided by the Control Heuristics depending on the analysis of the current state of the schedule.

Nevertheless, the opportunism in this approach remains limited in the sense that it typically requires scheduling an entire bottleneck (or at least a chunk of it) before being able to switch to another one. For this reason, such scheduling techniques are called macro-opportunistic scheduling. The scheduling technique has been further improved by a micro-opportunistic scheduling approach where instead of making macro-level decisions, each activity is made a decision point with its own measure of criticality.

2.5.3 MICRO-BOSS

MICRO-BOSS is a new approach to factory scheduling which simultaneously reduces tardiness and inventory costs (both finished-goods and in-process inventory costs). This scheduler was developed by Norman Sadeh [Sad91] in order to further increase
the performance of a scheduler compared to OPIS. Rather than scheduling entire bottleneck resources one by one, this approach focuses on the scheduling of critical operations. The scheduler accounts for a variety of costs including tardiness cost, in-process inventory costs, and finished-goods inventory costs and these costs are used to continuously update demand profiles that reflect contention between unscheduled jobs for the allocation of machines in function of time. By closely monitoring the evolution of bottlenecks during the construction of the schedule, and constantly
focusing on the scheduling of operations that rely most on the allocation of the most contended resource/time intervals, this scheduling approach has allowed for impressive reductions in inventory while maintaining tardiness at a very low level.

The micro-opportunistic search procedure is presented next:

1. If all operations have been scheduled then stop; else goto step 2;

2. Apply the consistency enforcing procedure;

3. Perform the look-ahead analysis: evaluate the reliance of each unscheduled operation on the availability of its remaining possible reservations, and measure resource contention over time.

4. Select the next operation to be scheduled (so-called operation ordering heuristic). In this, select the operation that relies most on the most contended (i.e., most highly relied upon) resource/time interval.

5. Select a promising reservation for that operation (so called reservation ordering heuristic).

6. Create a new search state by adding the new reservation assignment to the current partial schedule. Go back to step 1.

In this search procedure, the so-called opportunistic behavior results from the ability of the scheduler to constantly revise its search strategy and re-direct its effort towards the scheduling of the operation that appears to be the most critical in the current search state, critical operations are scheduled first. In reality, bottlenecks do not necessarily span over the entire scheduling horizon and in general they tend to shift before being entirely scheduled. A more flexible approach would allow to stop scheduling a resource as soon as another resource is identified as more constraining. This approach in which the evolution of bottlenecks is continuously monitored during the construction of the schedule and the problem solving effort redirected towards the most serious bottleneck is called a micro-opportunistic scheduling approach.
2.6 Other Non Jobshop Schedulers

Most of the discussion presented in this chapter is centered on jobshop schedulers which is the most understood scheduling problem. This thesis however tries to present a unified framework for developing a group of schedulers from diverse domains. Some of the single machine assignment results, such as scheduling the shortest task first or scheduling the bottleneck resources first, are applicable in the context of other schedulers. Other algorithms used in the context of jobshop and flowshop context as Johnson's algorithm are too specialized and specific and hence cannot be used in other scheduling domains. Since the thesis attempts to provide a general framework which encompasses a wide range of schedulers we have chosen a intelligent "generate and test" strategy for generating schedules.

Other schedulers of interest in this thesis include transportation schedulers, timetabling and process schedulers. The essential notion of transportation scheduling is to efficiently allocate transportation assets or resources (e.g., planes, ships, trucks) to some movements (e.g., bulk cargo, passengers) over time intervals in a manner so as to minimize the total transportation cost which is an indicator of the "goodness" of the schedule. During the last twenty years, many papers have been devoted to the development of optimization and approximation algorithms for vehicle routing and scheduling problems [GGJS57], [DR59],[FR76], [App71], [MH71], [Ron83]. The interest is due to the practical need for effective and efficient methods to handle physical distribution situations as well as to the intriguing nature of the underlying combinatorial optimization models. Some instances of transportation schedulers are tanker scheduling and truck dispatch scheduling problems.

The timetabling problem has been of interest since the mid 1960's [Bro64], [Col64], [MW66], [Bus67]. [SO74]. Various researchers have dealt with the problem in different ways the problem is of interest even now [FG89], [DMV89], [dW85] largely due to its varied nature, complexity and last, but not least, its large size. In simple words, the timetabling problem can be defined as scheduling of a certain number of sessions which must be attended by a specific group of enrollees, over a definite period of time, requiring certain resources. Some common examples of
timetabling problems are classroom scheduling, examination scheduling and conference scheduling. In classroom scheduling the sessions are classes which have to be attended by students who are the enrollees. The resources are teachers, rooms and teaching aids for the classroom to be held. Process scheduling deals with scheduling the activities in a process plant producing chemicals, petroleum and such products.

2.7 Handling Time in Schedulers

The representation of knowledge about time has attracted many researchers. Temporal reasoning is a large area in itself with applications in updating databases, automated planning, plan synthesis and natural language translation. Recent interest on Temporal Reasoning started when James Allen at Rochester New York, suggested in 1983-1984 that representing time knowledge using intervals rather than points of time, can be more more natural for may purposes in AI, he presented a formalism for reasoning about actions which is based on temporal logic [All83], [All84]. This allows a much wider range of actions to be described than with the previous approaches such as situation calculus. The original framework presented was quite general and could be used to characterize the different types of events, processes, actions and properties that can be described in simple english sentences.

Other significant work on temporal reasoning is presented by Allen and Pat Hayes where they put the interval system into the framework of first-order predicate, this work is contained in “A Commonsense Theory of Time” [AH85]. Ladkin and Maddux have also produced models of the interval calculus in “The Algebra of Convex Time Intervals” [LM87]. Marc Vilain and Henry Kautz have worked on the complexity of the interval calculus in “Constraint Propagation Algorithms for Temporal Reasoning” [VK86]. Significant work has been done in developing a logic-based calculus of events which is actually - expressing a calculus in a logic based language. The first notable attempt in that area was made by Kowalski and Sergot, the calculus is presented in their paper “A logic-based calculus of Events” [KS86]. In this framework the notion of event is taken to be more primitive than that of time and both are represented explicitly by means of Horn Clauses augmented with negation by failure. The main intended application is updating of databases and
narrative understanding, events are differentiated from times, as well as events which are partially ordered and concurrent.

Murray Shanahan in his paper Towards a Calculus for Temporal and Qualitative reasoning has presented a logic-based calculus for representing continuous change with some of the expressive power of the algebraic style of representation used in qualitative reasoning. This formalism is based on Kowalski and Sergot's event Calculus, which is well equipped to represent discrete change, but is not so good in representing continuous change.

Some of the important characteristics that are relevant to temporal reasoning in the area of scheduling are:

- The representation should allow significant imprecision. Much temporal knowledge is strictly relative (e.g. A is before B) and has little relation to absolute dates.

- The representation should allow uncertainty of information. Often the exact relationship between two times is not known, but some constraint which could be related are known.

- The representation should allow one to vary the grain of reasoning. For example when modeling jobshop scheduling one may need to consider time in terms of half hours but when modeling time in a ocean tanker scheduling the range of time could be in terms of days or even months.

Allen's temporal relations are shown in Figure 3-15 in Chapter 3 and an implementation based on his relations is presented in Section 3.5.

2.8 Conclusion

In the current chapter we have presented two perspectives on scheduling, namely the operations research and constraint satisfaction perspective for solving scheduling problems. The primary interest of this thesis is to develop a generic software
framework to help cut down on the development time of schedulers and provide a basic method for building new schedulers; we use concepts provided by both of these paradigms. Scheduling problems arise in many domains: in the manufacturing domain, tasks are referred to as jobs and correspond to parts or batches of parts that need to be processed on a set of machines. In hospitals often scheduling problems arise where tasks are patients, and resources are nurses, hospital beds or medical equipment required to treat the patients. Scheduling problems arise in schools, where the tasks are classes and resources can be teachers, classrooms and students.

The common theme in all these problems is how to identify task, resources and constraints and guide the allocation of resources to tasks in a systematic and intelligent manner. We explore some ways to answer this question in the following chapters.
Chapter 3

Design of a Scheduling Framework

"In the beginning was the word. But by the time the second word was added to it, there was trouble. For with it came syntax...."

- John Simon.

3.1 A Generic Scheduling Framework

This thesis presents a generic scheduling framework which makes it easier for scheduler developers to build customized scheduling applications, and provides end users a quick and easy means to test and implement new scheduling strategies. The scheduling framework is organized as a collection of scheduler clusters where each scheduler cluster represents a class of schedulers solving related problems. Jobshop, transportation, timetabling, and process plant scheduler clusters are examples of scheduler clusters and are presented in great detail in the following chapters.

Section 3.1.2 of this chapter describes our general approach to scheduling. The approach taken is knowledge based, allowing easy incorporation of heuristic knowledge of which tasks and resources to choose. The environment provides a partial classification of scheduling problems into clusters, as explained in Section 3.1.2. For the environment to be expandable and flexible, it must be straightforward to specify
scheduler clusters. For this purpose a metalanguage is provided and such a language is implemented in the system by writing DCG grammar rules. Examples of grammars and the style of specifying clusters and knowledge within clusters is given in Section 3.2.

Each scheduler is built around an iterative procedure. One iteration consists of choosing a task, choosing a resource, checking constraints, and propagating the effects of the allocation of the resource to the task for constructing the rest of the schedule. This basic mechanism is discussed in Section 3.3.

Loosely two approaches for computer generation of schedules have been suggested - an approach based on operations research (OR) and an approach based on artificial intelligence (AI). The OR approach involves formulating the problem in such a way so that known algorithms such as branch and bound and dynamic programming can be applied. Traditional OR methods seek to represent the generation machinery of schedulers within mathematically defined constraints, which is very useful for specific problems. However often in industry the scheduling problems addressed are too diverse and general for a mathematical solution.

3.1.1 How to start developing a scheduler?

The AI approach to scheduling attempts to solve the scheduling problem by intelligent guesswork, searching for promising combinations of resources to be allocated to tasks, satisfying constraints. AI approaches are willing to trade optimality for the time taken to obtain a solution. On small problems, there is little to choose between the methods as all reasonable schedules can be generated and an optimal one chosen. For larger problems, the AI approach allows for some experimentation with partial schedules which is useful for the dynamic nature of scheduling in industry. Flexibility in coping with dynamic industry needs has been a major factor in our design decisions.

The starting point for a generic scheduler was [FS91] which proposed to visualize a schedule as an enumeration tree, combining both AI and OR approaches. A more complete version of this work appears in [SS93]. At any stage of schedule generation, the tree represented a decomposition of the scheduling problem being solved. Nodes
Figure 3-1: Generic Scheduler Interface

at various depths represented problems that remained to be solved or solutions of subproblems. A particular problem would be decomposed using AI techniques into subproblems which may be instances of known OR problems. For example, we might have subproblems which might be solved by an assignment algorithm or a linear programming model.

The generic scheduler evolved into essentially a generate and test system which resides behind an extensive user interface. During schedule generation, resources
are allocated to tasks iteratively. Each iteration consists of choosing a task to be scheduled, choosing appropriate resources to be allocated, allocation of resources to tasks followed by constraint propagation which affects the remaining unscheduled tasks. This basic scheduling strategy is common to all schedulers developed in the environment. The philosophy of scheduling is presented in Figure 3-1.

We envisage the generic scheduling environment to be used for developing specific scheduler by two key players - the scheduler developer and the user. The scheduler developer tailors the basic scheduling algorithm presented later in this chapter in Figure 3-11. The problem domain and object types are designed by the scheduler developer and he/she may be also required to design a few predicates for implementing the current scheduler. The problem solving methodology in Figure 3-1 is partly determined by the scheduler developer who provides basic heuristics to be used during the scheduling process as well the operations research formulations to which the current problem can be reduced.

The user is the central figure in the environment shown in Figure 3-1. He/She provides the interface with the inputs required by the system and monitors the output provided by the generate and tester unit in the figure. The dotted arrows in Figure 3-1 denote information which is going into the system while solid arrows show information flowing out of the system. The information going in to the system in the form of program inputs are problem description data, problem constraints and predicates. The generate and tester unit produces the output in the form of a schedule, and depending on the output the user uses scheduling and testing judgment to either select the solution or modify the problem solving methodology by using a different heuristics or manually scheduling all the tasks. The concepts in Figure 3-1 are further presented in more detail in Figure 3-10.

### 3.1.2 Scheduler Clusters

Are all schedulers the same? Yes, according to the classical definition of scheduling from Baker [Bak74] which states that "scheduling is the allocation of resources to tasks subject to constraints". At some level of abstraction, all schedulers can be described by the scheduling strategy given in the previous Subsection 3.1.1. No,
in the sense that specialized methods have been developed to deal with special characteristics of scheduling problems in different domains. In order to allow a range of schedulers to exist in a generic scheduling framework, we classify schedulers.

![Scheduler Classification Diagram]

Figure 3-2: Scheduler Classification and Specific Scheduler Generation

The classification of schedulers that we have developed is shown in Figure 3-2. Some scheduler clusters which have been designed and are shown in the figure
are jobshop schedulers which assign resources to tasks for periods of time subject to constraints such as precedence constraints between tasks, transportation schedulers which deliver a resource from several locations to several destinations, and timetabling schedulers for allocating venues to sessions such that the cost of allocation or the number of conflicts in the system are minimized. The remaining arrow in the Figure 3-2 indicates that the three clusters do not cover all schedulers, and one another instance of process scheduling is also presented in this thesis.

The four scheduler clusters have a similar core part, consisting of the basic attributes needed for the schedulers and useful predicates implementing central concepts. The core part has to be customized for each scheduler as explained in later sections of this chapter. Particular schedulers are built by customizing and instantiating the scheduler clusters. For example, the ocean tanker scheduler or truck dispatch scheduler are instances of a transportation scheduler and are built by extensions and customizations done to the transportation scheduler cluster.

3.2 Specification of Scheduler Clusters

In the current section we present the different entities which constitute individual scheduling clusters. The discussion presented in this section uses examples based on two clusters the jobshop and transportation clusters which have been presented in greater detail in Chapters 4 and 5. A scheduler cluster consists of two parts. The first part is a grammar specifying the form of input accepted by each scheduler within the cluster. The grammar determines the syntax for tasks, resources, constraints, schedule heuristics, schedule parameters, costs, and objective functions. A language has been developed for expressing both the form of knowledge needed for each scheduler cluster and the input form of that knowledge. Grammars defining the languages have been implemented using definite clause grammars (DCGs) which are a notational variant of a class of Prolog programs, and are executable. The second part is the scheduling engine which has some code stubs. The scheduling engine will be described in Section 3.3.

Every scheduler cluster must specify at least 7 classes of items. These are the tasks, resources, constraints, schedule heuristics, schedule costs, schedule parameters
and desired objective function to be optimized by the schedule. DCGs provide a very convenient way of expressing the items. A grammar rule used in the environment for parsing scheduler specification is given in the right hand half of Figure 3.2. It is intended to speak for itself by clearly stating the items that must appear in the scheduler specification.

% - Parse Generator in Prolog-
grammar(InputFile, Dataset, Tasks, Resources, Constraints, Heuristics, Costs, SchedParams, Objective)
  ___ read_file(InputFile, Tokens),
  phrase(scheduler(Dataset, Tasks, Resources, Constraints, Heuristics, Costs, SchedParams, Objective), Tokens).

% Common rule for all scheduling clusters
scheduler(Dataset, Tasks, Resources, Constraints, Heuristics, Costs, SchedParams, Objective)
  ___ dataset(Dataset),
      tasks(Tasks),
      resources(Resources),
      constraints(Constraints),
      schedule.heuristics(Heuristics),
      costs(Costs),
      schedule.parameters(SchedParams),
      objective(Objective).

Figure 3-3: Recognizing Schedule Input

The grammar defining a scheduler cluster is used to parse scheduler input and generate an intermediate form consisting of Prolog facts. The top level predicate used in generating the intermediate Prolog form is grammar/9 which reads the input-file InputFile and uses the Prolog predicate phrase/2 to collect various schedule information. The grammar rule is given in the left half of Figure 3.2.

Each predicate in the body of the rule for scheduler/8, namely dataset, tasks, resources, constraints, schedule.heuristics, costs, schedule.parameters, and objective, has a clear definition in a similar style, which we discuss in turn. Several of the items distinguish between mandatory fields which are values which must appear in the scheduler specification and extra fields which are optional and allow the user to introduce problem specific fields.

3.2.1 Task Specification

The tasks in different schedulers depend on the nature of the problem being solved, identifying the task fields is one of the important things which should be done before the design of a cluster. Let us have a birds view of what a task really is under these two different scheduling clusters.
The task specification for a scheduler which is an instance of the jobshop cluster consists of four mandatory fields: a unique task identifier, a descriptive name, the earliest start time of the task, and the priority of the task.

```
% Top Level task specification for the jobshop cluster

(task(task([task.id::TaskId, task.name::TaskName, est::EST, priority::Priority[ExtraFields]]))
  taskId(TaskId),[''],
  task_name(TaskName),[''],
  est(EST),[''],
  priority(Priority),[''],
  #,
  extra_fields(ExtraFields),
  #.

% Top Level task specification for the transportation zone.

(task(task([taskId::TaskId, addr::Address, zone::Zone, service_stations::Stations, qlow::QLow, qhigh::QHigh, w1::W1, w2::W2[ExtraFields]]))
  taskId(TaskId),[''],
  address(Address),[''],
  zone(Zone),[''],
  service(Stations),[''],
  quantity(QLow,QHigh),[''],
  delivery_window(W1,W2),[''],
  #,
  extra_fields(ExtraFields)
  #.
```

Figure 3-4: DCG Recognizing Tasks

The task specification for a scheduler which is an instance of the transportation cluster consists of six mandatory fields: a unique delivery identifier, address of delivery, a zone of delivery (a zone here means a set of delivery ports which are proximal to each other), supply service stations, the quantity to be delivered specified by a minimum and a maximum required, and temporal information in the form of a window of delivery which states the earliest and latest allowable times of delivery. Sample rules for collecting task information are presented in Figure 3-4.

The grammar for recognizing tasks in the jobshop scheduling cluster is presented in Section 4.5.1 while the grammar for recognizing tasks in a transportation scheduler is presented in Section 5.5.1 in more detail. The specification of fields in a scheduler cluster is extensible. There is an explicit provision for including extra fields after the standard fields in both task specifications and resource specifications. In our implementation, the extra fields are enclosed within the ‘#’ character. Each of the extra fields are stored as shown in Figure 3-4, 3-5.

For example the task extra fields priority:3, order:1 are stored as user_defined::priority:3 and user_defined::order::1. The extra fields can also be used to define new schedule heuristics, where the precise order of task selection and resource
ordering can be determined by defining additional fields. This was used in experimenting with different scheduling heuristics for the carshop and transportation schedulers.

### 3.2.2 Resource Specification

Resource specification identifying each resource is the next thing which has to be done for each cluster, this is done in a manner similar to task specification. The resource specification for instances of the jobshop cluster consists of five mandatory fields. They are a unique resource identifier, a load pertaining to the current utilization of the resource, the number of hours the resource is currently allocated, the skill level of the resource, and temporal information specifying the interval within which the resource can be allocated.

The resource specification for instances of the transportation cluster is very similar and consists of a resource identifier, size, capacity of the resource. Constraints such as loading constraints and other cluster specific information are also specified in each resource instance. A sample rule for collecting resource information is presented in Figure 3-6. Details of resource specification in the case of jobshop scheduler cluster are presented in Section 4.5.2 and for the case of transportation scheduler cluster are presented in Section 5.5.2.

Note that we allow extra fields which are handled as described for task specification.
3.2.3 Constraint Specification

Allocation of resources to tasks proceeds subject to constraints which reduce the search space by forbidding assignments. Constraints in the scheduling problem can be categorized into several types some of which are temporal, calendar, resource, capacity, and process constraints. Temporal constraints relate the time points of activities, calendar constraints relate a preference calendar with the time points of activities, resource constraints relate resource assignment with a task. Capacity constraints relate the amount of a resource used by a task with a resource assignment, process constraints relate the alternative activities of a process. There always exist very specific constraints with respect to the clusters which are being developed and these are not inherited from the scheduler clusters. We present some specific examples of two important constraint types occurring in the scheduler clusters.

Example Temporal Constraints: All jobs in the jobshop schedulers must be started and completed between start/end time slots. Furthermore, a task must be completed before its due time if provided as input. Precedence relations are other temporal constraints in the system which force certain tasks to be scheduled before others. In transportation schedulers the deliveries to specific destinations need to be made within specific time windows, also resources available for new deliveries begin only at specific times since current deliveries might be underway. In the time tabling cluster, sessions are restricted to begin
only after a certain period during the day.

Example Resource Constraints: Resource constraints curtail the allocation of resource combinations to tasks. For the jobshop scheduler cluster one resource constraint forbids an operator from working on certain tasks while another resource constraint forbids a certain set of operators from working together. Similarly infeasible resource allocation constraints prevent certain transport vessels from being assigned to serve specific destinations, also similar constraints in the time tabling schedulers prevent the assignment of certain sessions to specific venues.

| %--- dcg recognizing constraints --- |
| constraints(Constraints) → |
| constraints(Constraints), |
| task_constraints(Task.Cons), |
| resource_constraints(Rcs.Cons), |
| temporal_constraints(Temp.Cons), |
| { append(Task.Cons, Rcs.Cons, Cons), |
| append(Cons, Temp.Cons, Constraints) } | |
| constraints([]) → []. |

<p>| %--- simplified temporal constraint for |
| jobshop scheduler |
| temporal_constraints(Prev.Cons) --- |</p>
<table>
<thead>
<tr>
<th>prec_constraints(Prev.Cons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>['precedes''],</td>
</tr>
<tr>
<td>[TaskId],</td>
</tr>
<tr>
<td>[Task,''],</td>
</tr>
<tr>
<td>preceding_tasks(Tasks),</td>
</tr>
<tr>
<td>['',''],</td>
</tr>
<tr>
<td>prec_constraints(Prev.Precs),</td>
</tr>
<tr>
<td>{Prev.Cons=[precedes(TaskId, Task, Tasks)</td>
</tr>
<tr>
<td>prec_constraints([])</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>[].</td>
</tr>
</tbody>
</table>

Some of the constraint types like capacity constraints are subsumed by the resource/task constraints.

3.2.4 Schedule Heuristics

Heuristics in the system are categorized into pre-defined and user-defined heuristics. Each scheduler is guided by schedule heuristics which determine the next task to be scheduled and the resource to be allocated to the scheduled task.

Pre-Defined Heuristics

Certain heuristics are pre-defined in the system for both order of task selection and resource allocation. Examples of some common pre-defined task selection heuristics
are *Weighted Shortest Task First* and *Weighted Longest Task First* in the case of jobshop schedulers, while *Largest Delivery First* and *Farthest Task first* are pre-defined heuristics for transportation schedulers.

Examples of pre-defined resources selection for the jobshop scheduler are *Earliest Finishing Operators, Minimizing Operator Workload*, while examples of some pre-defined resource selection heuristics for the Transportation schedulers are *Highest Resource Capacity First* and *Most Constrained Vessel First*. The pre-defined heuristics provide the user a convenient way of running the scheduler and running simulations to see scheduling results.

```
schedule_heuristics::  
  Task_Heuristics : user_defined;
  Heuristic_Id : 1 ;
  Operations :
    
    [  
      Operation : sort  
        on Primary (user_defined.order) asc;
    ];

  Task_Heuristics : user_defined;
  Heuristic_Id : 2 ;
  Operations :
    
    [  
      Operation : select;  
        Conditions : [(user_defined.priority),2,3];  
      Operation : sort  
        on Primary (standard.qhigh) asc;
    ];  
    [Operation : choose_remaining];

  Task_Heuristics : pre_defined;
  Heuristic_Id : 3 ;
  Operation : highest_demand_first;
  Task_Hs_Execution_Order: [2,3,1];

Resource_Heuristics : user_defined;
Heuristic_Id : 2 ;
Operations :

  [  
    Operation : sort  
      on Primary (user_defined.order) asc;
  ];

Resource_Heuristics : user_defined;
Heuristic_Id : 1 ;
Operations :

  [  
    Operation : select;  
      Conditions : [(standard.capacity),2,45];  
      Operation : sort  
        on Primary (standard.w1) asc;
    ];  
    [Operation : choose_remaining];

Resource_Hs_Execution_Order: [2,1];
```

Figure 3-8: Example Heuristic Specification for a Transportation Scheduler

**User-Defined Heuristics**

A simple language is provided to the user to allow him/her to specify and control the order in which the tasks are scheduled. The heuristics are expressed in a simple
language consisting of sequences of instructions of one of three forms. The first form whose operation is select, is meant to capture the select operation from SQL, while the second form whose operation is sort captures the sort operation from SQL. The third form choose_remaining appends the remain tasks to the end of the schedulable task list. Aggregate operators like max, min are also provided while the most common aggregate operator used is sort. Sort sorts the input according to a specified primary and an optional secondary key.

Part of the grammar for recognizing schedule heuristics is presented in Figure 3-9. This shows how the heuristics in the system are parsed. Each of the task and resource heuristics are recognized by the ordering_hs rule which expects a series of instructions.

Sample user-defined heuristics have been shown in Figure 3-8. The first task heuristic sorts all the given tasks based on an extra field order. The user can precisely control the order of task selection by defining this extra field and numbering the extra fields 1, 2, ... where the order signifies the order in which the tasks will be scheduled. The exact ordering of tasks is achieved by sorting the tasks, using order as the primary key. The second user-defined heuristic selects tasks which have a priority greater than 3, and further schedules the task with the highest “maximum demand” first (qhigh is the maximum demand at a port). Remaining tasks with priorities less than 3 are scheduled later as specified by the instruction Operation: choose_remaining.

3.2.5 Heuristic Execution

Task and resource heuristic execution order are specified in the input. In the example as indicated in Figure 3-8 the heuristic orders of task and resources are specified in the fields Task_Hs_Execution_Order: [2, 3, 1] and Resource_Hs_Execution_Order: [2, 1] respectively. During schedule execution the first pair of heuristics executed are <2, 2> for task and resource i.e. task heuristic with identifier 2 and resource heuristic with identifier 2. This is followed by the second pair of heuristics <2, 1> followed by the remaining items in the cartesian product of [2, 3, 1] and [2, 1] in order. The best schedule from those generated is reported.
We feel that by providing a mechanism for customizing the control of the scheduler, a flexible scheduler has been developed, in contrast to the schedulers existing in industry.

A part of the grammar for recognizing the ordering heuristics is presented in Figure 3-9.

```
ordering_hs(InstructionList) ->
  instruction(Instr), [;:],
ordering_hs(RestInstr),
InstructionList=[Instr|RestInstr].
ordering_hs([Instruction]) ->
  instruction([Instruction]).
```

```
instruction([choose_remaining]) ->
  ['operation', ';', 'choose_remaining', ';', ']').
instruction([select, Conditions, sort, PrimarySortFld, SecondarySortFld]) ->
  ['operation', ';', 'select', ';', 'conditions', Conditions, ';', 'sort', Instruction(PrimarySortFld, SecondarySortFld), ';'],
  instruction([sort, PrimarySortFld, SecondarySortFld]) ->
  ['operation', 'sort', Instruction(PrimarySortFld, SecondarySortFld), ';'],
instruction([]) -> [].
```

Figure 3-9: Grammar For Recognizing Heuristic Specification

### 3.2.6 Schedule Parameters

Each scheduler cluster has concepts that pertain only to schedulers which are instances of that cluster, and not schedulers lying outside the cluster. Cluster-specific information is given as schedule parameters.

For example, the transportation cluster needs to know about distances between different service ports and destinations. The distance information can be represented as a graph. Our implementation uses a connectivity matrix. The other parameter for the transportation cluster is the maximum number of deliveries which can be made by a resource during a single trip.

Two schedule parameters for the jobshop cluster need to be specified. The first is heuristic weights for each task selection heuristic. For example in the car shop scheduler tasks like brake jobs, fix lights, tuneup, could have weights 5.0, 4.0 and 1.0 according to the weighted shortest task first heuristic while the weights could be 1.0, 3.0 and 5.0 according to the weighted longest task first heuristic. Tasks are not dispatched purely based on heuristic weights but also based on temporal information. Tasks whose earliest start times lie within a specified range compete for dispatch and this time window is called the dispatch contention window. The second parameter
that must be specified is the size of the dispatch contention window. Concepts relating to dispatch contention window are presented in more detail in Section 4.7.1.

3.2.7 **Objective Specification and Costs**

In each scheduler every assignment of resource to task for a finite period of time gives rise to an assignment cost. The cost of a schedule is an aggregate of several assignment costs. Costs arise in almost all scheduling clusters, and minimizing costs is most often the most common requirement in all scheduling clusters. The cost function in the transportation scheduler is the sum of maintenance costs and cost of actual travel per unit distance for each category of transportation vessel. In the jobshop scheduler each resource has a fixed cost of operation for each unit of time assigned and the cost of the schedule is calculated based on actual time of allocation of different resources.

The objective considered are *minimizing cost of schedule*, where costs arise due to assignment of resources to tasks. Other objectives could be *minimizing total completion time (makespan)* of the schedule or balancing operator loads in the case of jobshop schedulers. Multiple objectives can be specified in the scheduler and the post analyzer of the scheduler using statistics generated by the post analyzer.

3.3 **Schedule Generation**

A user wishing to generate a schedule must give the specific tasks to be scheduled, the resources available and any constraints. As discussed in the previous section, this information is parsed by the grammar, and compiled into an intermediate form that can be used by the scheduling engine. This section discusses how the scheduler runs and uses the heuristic knowledge that has been specified.

3.3.1 **The Scheduling Engine**

The scheduler has three modules which are depicted in Figure 3-10 giving an overview of the scheduler. The schedule pre-analyzer checks the partial feasibility of a schedule and can point out obvious anomalies. The schedule post-analyzer selects the best schedule that has been generated according to the specified cost function, and also computes useful statistics such as the make-span for jobshop schedulers.
The core of the scheduler consists of a task dispatcher, resource allocator and constraint propagator. Control of the scheduler is provided by the heuristics selector, which guides the schedule by determining the task selection and resource selection heuristics. We review each in turn.

**Task Dispatcher:** This determines the order in which different tasks are dispatched in the system. Before the scheduler starts, the tasks are ordered according to the specified task ordering heuristics. As the scheduler runs, tasks are selected for dispatch according to this order. The task dispatcher makes sure that each dispatched
task satisfies all temporal constraints before being passed to the resource allocator.

**Resource Allocator:** The resource allocator allocates resources to a schedulable task. Since a number of resource-task allocation combinations exist for each task, the resource allocator is guided by resource selection heuristics to determine resource selection for each task. The resource allocator also determines the start time of each task, as it depends both on the temporal constraints on the task, and the periods of availability of the resource.

**Constraint Propagator:** Once a task is allocated resources and has been temporally placed along the time line, the constraint propagator propagates constraints in the system. The constraint propagator determines the next set of tasks which contend for dispatch, and the earliest start times are determined by the constraints in the system.

**Heuristics Selector:** The heuristic selector provides flow of control information to the task dispatcher and resource allocator units. The heuristics selector can use a user-defined heuristic, or one from the set of standard heuristics defined for each scheduling cluster in the system. Task dispatch ordering is determined by the task selection heuristic, while resource allocation is guided by using resource selection heuristics. Pre-defined task selection heuristics in jobshop schedulers associate dispatch weights with all the tasks which need to be scheduled. In the transportation schedulers tasks are sorted based on one of the attributes of the task object. Resource ordering heuristics also order resources by one of their attributes and the task and resource ordering is updated at the end of each schedule iteration.

### 3.4 Schedule Generation

In this section we flesh out the schedule generation described in Figure 3-10. The Pre-Processor/Pre-Analyzer analyses the basic input data for inconsistencies, for example, in the case of the carshop scheduler the pre-analyzer does a partial feasibility check by verifying if there are sufficient resource hours available compared to the hours required by input tasks. The pre-processor tailors input data in the object form to be used by the scheduler, it also construct other useful structures
from the input data - like the conflict matrix for time tabling schedulers.

The algorithm presented in Figure 3-11 gives the essence of the schedulers developed under the generalized scheduler framework, it assumes that the pre-analysis/pre-processing is already done on the input data. The actual implementations of the algorithm are customized for specific schedulers with minor variations as mentioned in Figure 3-2. "Routing" is an example of a problem specific calculation perform in Step 5 in a transportation scheduler. Task and resource selection are done based on the current task/resource selection heuristics. Only those tasks which satisfy all current constraints are dispatched in step 1, resource constraints are checked at step 2. Step 3 updates resource capacities, resource availability and the current tasklist based on the task/resource pair selected. Propagation of constraints takes place in Step 4, and the schedulable intervals of all other tasks are modified depending on the time interval during which the current task was scheduled. The best schedule in a fixed number of iterations is reported to the user.

Implementing the Engine

The top level Prolog code for a specific scheduler is presented in Figure 3-12.

The scheduler gets input stored in the intermediate Prolog object form. The flow of control is controlled at two levels which are the inner and outer loops. The inner loop is generate_schedules/4, which generates each of the individual schedules based on a specific task, resource heuristic combinations used for a set of schedule iterations. The predicate post_analyze/3 implements the flow control shown at the bottom right hand side of Figure 3-10. When print is the first argument of the predicate, the predicate prints out the current schedule, else it generates schedule statistics and evaluates the current schedule. evaluate_and_update/3 evaluates the current schedule and the best schedule currently is asserted into the Prolog database.

Since the basic scheduling algorithm is of a generate-and-test the scheduler is monitored by next_schedule/1 which allows the scheduler to try a fixed number of iterations, the best schedule generated using the current set of heuristics is reported and control passes over to order_by_heuristic/1. Other examples of Prolog programs using generate-and-test to solving problems can be found in [SS86].
Input: Sets R(AvailableResources) and T(TaskList), C(Constraints), N(MaxSolutionsToTry)
Output: "Good" Feasible Schedule $S$

**Step 1. Task Selection:**
- If $T$ is empty
  - goto Step 5./*end of current schedule*/
- Select $T_i \in T$ depending on chosen Task Selection heuristic.

**Step 2. Resource Selection:**
- Select a resource $R_i \in R$, depending on chosen Resource Selection heuristic subject to resource constraints.
  1. Test for Capacity Constraints.
  2. Test for Availability Constraints.
  3. Avoid incompatible task-resource combination.
- If no $R_i$ can be selected
  - Report Unscheduleable task $T_i$ and goto Step 6.

**Temporal Satisfaction:**
- The following temporal constraint has to be satisfied:
  - $(ST_{i}, ET_{i}) \in T_i, SchedulableInterval$ and $(ET_{i}, ST_{i}) = T_i, Duration$
  - $(ST_{i}, ET_{i}) \in R_i, Availability$

**Step 3. Allocation:**
- Resource Allocation: Assign $R_i$ to $T_i$ where start time of allocation $ST_{i} \in T_i, SchedulableInterval$.
- TaskList Update: if completed fully, Remove $T_i$ from TaskList $T_i$ else update task $T_i$ suitably.
- ResourceList Update: Remove interval $(ST_{i}, ET_{i})$ from $R_i, Availability$.
- $T_i$ is the start and end time of task $T_i$.
- If $(R_i, Availability == Nil)$ then remove $R_i$ from ResourceList.

**Step 4. Constraint Propagation:**
- $\forall$ Tasks $T_j$ such that conflicts($T_i, T_j$) remove interval $(ST_{j}, ET_{j})$ from $T_j, SchedulableInterval$.
- Modify $EST_j$ and hence also $T_j, SchedulableInterval$ of all tasks $T_j$ which needing modification.
- Goto Step 1.

**Step 5. Calculations/Postprocessing:**
- Specific Calculations: Perform cluster specific calculations.
- Evaluation: Evaluate the goodness of the schedule.
- Compare goodness of current schedule and store $S$ if better solution.

**Step 6.**
- if ($\#$SchedulesGenerated > N)
  - goto Step 7
- else generate new schedule using problem specific control flow.

**Step 7. Schedule Output:**
- If no schedule generated
  - Print No Schedule Feasible
- else
  - Print best solution $S$ generated.
  - Stop.

---

**Figure 3-11:** "Generate and Test" Scheduling Algorithm

**Backtracking:** Alternative schedules are determined by *backtracking* which is one of the features making Prolog particularly interesting as a language of choice in
Figure 3-12: Top Level Prolog Scheduling Predicate

designing schedulers. The degree of backtracking can be controlled in each scheduler
cluster, for the jobshop schedulers the scheduler jumps back to the beginning of the schedule, while in the case of the transportation schedulers the scheduler does standard Prolog backtracking. Both these modes of backtracking are illustrated in the code presented in Figure 3-12.

Each scheduler is guided by schedule heuristics which determine the next task to be scheduled and the resource to be allocated to the scheduled task. The main scheduler cycle has been presented in Figure 3-12. Note that choose_task and choose_update_resources need more information about making their choice. Figure 3-10 illustrates the function of the heuristics selector which provides the control information to the basic cycle.

This description concludes the description of the basic scheduling philosophy in this thesis. All clusters are described in more detail in Chapters 4, 5 and 6 and 7 which also present some examples of schedules generated by the system. The next important issue is that of handling time in the generic scheduler framework, after which the representation of objects in the system is presented.
3.5 Handling Time in the Generic Framework

The representation of knowledge about time has attracted many researchers and a brief introduction to temporal representation has been presented in Section 2.7 in Chapter 2. One of the dimensions of any schedule is time and in this section a method for representing and updating temporal information in the generic scheduler framework is presented. In the current section we show how concepts developed in temporal reasoning can also be used in handling temporal constraints in scheduling by developing two predicates in Prolog which can be used in any schedule cluster. These are intersect_time_interval/4 which returns all successive intersections between two lists of time intervals, and update_time_interval/3 which removes a time interval from a list of time intervals.

In the current thesis the two predicates mentioned earlier have been used in the timetabling schedule cluster, while all other clusters use the same interval representation of time, but not the exact predicates defined here for handling temporal propagation. This is due to the reason that both jobshop and transportation clusters were developed earlier than the timetabling scheduler at about which time we decided to develop a unified framework for temporal representation.

An interval is a finite length of time which starts and ends at definite points. A time interval can be visually represented as a horizontal line with time going from left to right. A complete algorithm is presented in Allen’s paper on maintaining knowledge about temporal intervals [All84] where an interval based temporal logic is introduced, together with a computationally effective reasoning algorithm based on constraint propagation. This system is notable in offering a delicate balance between the expressive power and the efficiency of its deductive engine. We use only a part of the expressiveness of Allen’s work, the two main ideas used are expressing time as intervals, and using one of the Allen’s relations for propagating temporal constraints shown in Figure 3-15.

We can use time intervals to represent the periods of time over which tasks are performed. For example, if a job needs the following machining operations - a lathe operation first, followed by drilling, followed by burnishing then the time interval during which these tasks are performed can be drawn as shown in the first part of
Figure 3-14. Suppose some other operation may be needed in between the lathe and burnishing operation then the picture of the time interval looks like the second half of Figure 3-14, now there is a gap between those two intervals.

![Diagram of lathe, drilling, and burnishing operations with a gap between them.]

Figure 3-14: Example Time Representation of Events

![Diagram of Allen's Temporal Relations: X Before Y, X Same Y, X Overlaps Y, X During Y, X Starts Y, X Ends Y, X Meets Y.]

Figure 3-15: Allen's Temporal Relations

By providing a formal definition of these pictures of intervals we can say something about how our tasks are related in time. These pictures are described in terms
of end-points of intervals, and the end-points of an interval $I$ can be represented as $lb_i$ and $ub_i$ where $lb$ and $ub$ are the lower and upper bound of the interval. The constraint that $lb_i < ub_i$ is assumed and this helps us to reason with the endpoints alone, using just points and not mentioning intervals.

Suppose a task has to be scheduled during the interval specified in the set $[(2,5),(6,7),(9,12)]$ and the resource is available only in the interval $[(4,6),(8,14)]$, and the duration of the task is 2 units, the question posed to the system is when can the task be scheduled? assuming there were no other constraints. In this case we find the intersection between the two list of time intervals using the `intersect_time_interval` predicate whose arguments are Interval1, Interval2, Duration, Intersection. The first such intersection occurs in the time range $(9,11)$, while another intersections is $(10,12)$. If the resource were allocated from $(9,11)$ then constraint propagation is used to update the time interval during which the resource is now available i.e. $[(4,6),(8,9),(11,14)]$, its arguments are Removed Interval, Current Interval and Updated Interval.

We present the implementation of Prolog temporal predicates for finding intersecting intervals in Figure 3-16. These correspond to finding intersections between intervals which are related to each other in a manner specified by one of the 13 Allen’s relations.

The first 7 relations are shown in Figure 3-15, each of these relations has an inverse excepting (X Same Y). The first predicate deals with the case (X Before Y) or (X Meets Y) two of the 13 Allen’s relations. In this case it is necessary to find the intersection between time interval lists $[(T1,T2) | R1]$ and $[(T3,T4) | R2]$, since the interval $(T1,T2)$ before $(T3,T4)$ is true, it is necessary to find the intersection now between $R1$ and $[(T3,T4) | R2]$. Backtracking is used to generate all possible intersections and can be seen in the predicate sample shown in Figure 3-16.

The other predicate developed for temporal propagation is shown in Figure 3-17. The arguments are Removed Interval, Current Interval and Updated Interval. Some of Allens relations cannot occur in this case and have been indicated in the predicate list shown in Figure 3-17. The first situation (X Before Y) or (X Meets Y) is not possible because we make the assumption that the removed interval lies
% All Allen's Relations
% <X-> <Y->
% <T1-X-T2> <T3-Y-T4>
% (X Before Y) or (X meets Y).
intersect.time.interval([[T1,T2],[R1],[T3,T4],[R2],Dur,I]) ->
T2 <= T3,
]
intersect.time.interval([[T1,T2],[R1],[T3,T4],[R2],Dur,I]).
% Inverse Relation
% <T3-Y-T4> <T1-X-T2>
% <T3-Y-T4> <T1-X-T2>
% (Y Before X) and (Y meets X).
intersect.time.interval([[T1,T2],[R1],[T3,T4],[R2],Dur,I]) ->
T4 <= T1,
]
intersect.time.interval([[T1,T2],[R1],[R2],Dur,I]).
% X equals Y.
% <T1—X—T2>
% <T1—Y—T2>
intersect.time.interval([[T1,T2],[[,]],[T1,T2],[[,]],Dur,I]) ->
T2 - T1 > Dur,
 Temp is T1+Dur,
 I = (T1,Temp).
intersect.time.interval([[T1,T2],[R1],[T1,T2],[R2],Dur,I]) ->
T2 - T1 <= Dur,
]
intersect.time.interval([[R1,R2,Dur,I]).

% X ends Y.
% <T1—X—T2>
% <T1—Y—T2>
intersect.time.interval([[T1,T2],[[,]],[T1,T4],[[,]],Dur,I]) ->
T4 > T2,
 T2 - T1 <= Dur,
 I = (T1,High).
intersect.time.interval([[T1,T2],[R1],[T1,T4],[R2],Dur,I]) ->
T4 > T2,
 T2 - T1 <= Dur,
]
intersect.time.interval([[Temp,T4],[R1],[Temp,T2],[R2],Dur,I]).
% X starts Y.
% <T1—X—T2>
% <T1—Y—T4>
intersect.time.interval([[T1,T2],[[,]],[T1,T4],[[,]],Dur,I]) ->
T4 > T2,
 T2 - T1 <= Dur,
]
intersect.time.interval([[T1,T2],[R1],[T1,T4],[R2],Dur,I]) ->
T4 > T2,
 T2 - T1 <= Dur,
]
intersect.time.interval([[Temp,T4],[R1],[Temp,T2],[R2],Dur,I]).
% Y starts X.
% <T1—X—T2>
% <T3—Y—T2>
intersect.time.interval([[T1,T2],[[,]],[T3,T2],[[,]],Dur,I]) ->
T3 < T1,
 T2 - T1 <= Dur,
 Temp is T3+Dur,
 I = (T3,Temp).
intersect.time.interval([[T1,T2],[R1],[T3,T2],[R2],Dur,I]) ->
T3 < T1,
 T2 - T1 <= Dur,
]
intersect.time.interval([[R1,R2,Dur,I]).

% Variable I is used for Intersection
intersect.time.interval([[T1,T2],[[,]],[T1,T4],[[,]],Dur,I]) ->
T2 > T4,
 T4 - T1 > Dur,
 High is T1+Dur,
 I = (T1,High).
intersect.time.interval([[T1,T2],[R1],[T1,T4],[R2],Dur,I]) ->
T2 > T4,
 T4 - T1 <= Dur,
]
Temp is T1+Dur,
 intersect.time.interval([[Temp,T2],[R1],[R2],Dur,I]).
intersect.time.interval([[T1,T2],[R1],[T1,T4],[R2],Dur,I]) ->
T2 > T4,
 T4 - T1 > Dur,
]
Temp is T1+1, % shift by one unit
 intersect.time.interval([[Temp,T2],[R1],[[Temp,T4],[R2],Dur,I]).

% Y ends X.
% <T1—Y—T2>
% <T3—Y—T2>
intersect.time.interval([[T1,T2],[[,]],[T3,T2],[[,]],Dur,I]) ->
T3 < T1,
 T2 - T1 <= Dur,
]
intersect.time.interval([[T1,T2],[R1],[T3,T2],[R2],Dur,I]) ->
T3 < T1,
 T2 - T1 <= Dur,
]
intersect.time.interval([[Temp,T2],[R1],[[Temp,T4],[R2],Dur,I]).

Figure 3-16: Some Prolog Temporal Predicates for Finding Intersecting Intervals
% All Allen's Cases are taken care of —
update_time_interval([T1..T2], [], []) :- !.
% <X-> <Y->
% <T1-X-T2> <T3-Y-T4>
% (X Before Y) or (X meets Y).
% Is not possible.
% Inverse Relation
% <T3-Y-T4> <T1-X-T2>
% <T3-Y-T4> <T1-X-T2>
% (Y Before X) or (Y meets X).
update_time_interval([T1,T2],[T3,T4],[R2],
                   [(T3,T4)][UpdatedInterval]) :-
    T4 =< T1, !,
    update_time_interval([T1,T2],[R2],[UpdatedInterval]).
% X equals.
% <T1—X—T2>
% <T1—Y—T2>
update_time_interval([T1,T2],[T1,T2],[R2],[R2]) :- !.
% X starts Y
% <T1—X—T2>
% <T1—Y—T4>
update_time_interval([T1,T2],[T1,T4],[R2],[T2,T4],[R2]) :-
    T4 > T2, !.
% Y starts X
% <T1—X—T2>
% <T1—Y—T4>
update_time_interval([T1,T3],[T3,T4],[R2],[T2,T4],[R2]) :-
    T3 < T1, T2 < T4, !.
% Y ends X.
% <T1—X—T2>
% <T3—Y—T2>
update_time_interval([T1,T2],[T3,T2],[R2],
                    [(T3,T1)][R2]) :-
    T3 < T1, !.
% Y during X.
% <T1—X—T2>
% <T3—Y—T4>
% Situation Does not Occur.
% <T1—X—T2> Y overlaps X.
% <T3—Y—T4>
% Situation Does not Occur.

Figure 3-17: Temporal Predicates in Prolog for Updating Intervals

in the interval from which it is being removed. In case of certain intervals as in the
case of (Y Before X) and (Y meets X) finding the intersection amount to finding
intersections between time intervals [(T1,T2) | R1] and R2.

3.6 The Object System

The framework takes an object-oriented approach using ideas from Minsky's seminal
paper on frames [Min75]. Each object is represented as a single arity compound
term, whose only argument is a list of object attributes. For example, a resource with
3 attributes resource identifier (rcsId), available time interval (avail_interval),
skill level of the resource (skill_level) is represented as resource([rcsId::Id, avail_interval:: Av.Interval, skill_level:: SkillLevel]) which is a compound term of arity 1 with the first argument being a list of object identifier and object value pairs. An alternative representation of the resource could be resource(Id, Av.Interval, SkillLevel) which is a compound term of arity 3. Such a representation is not used since fixed arity compound terms make customization and maintenance of schedulers difficult. Similar representation of frames have been done in Lisp using nested association lists. These are a list of elements, each of which can be identified and extracted from the list by using its so-called key.

The object system helped in making the system more generic and has very low overhead since it only implements the necessary parts of a large true object oriented system. Some of the predicates used for making the objects are presented in Figure 3-18, 3-19. The functions are a simplified version of those defined in FRL, the frame representation language designed by Bruce Roberts and Ira Goldstein [RP77].

Making a new object:

make_object(ObjName, Attribs, Object) makes a new object in the system. Attributes of the object are maintained as a a list. Making a new object is similar to a series of DEFPROP statements in Frame Representation Language(FRL), and make_object initializes all attributes of the object to default values. If the values of some attributes have to be pre-initialized to known values then the predicate initialize_Attribs is provided with a list of tuples of the form (Attrib::Val), else the values are initialized to an empty list.

Accessing attribute values:

access.Attr_val(ListOfAttribs::Object,ListOfVals) accesses the values of a list of attributes of a given object 'Object' and returns the vaules in ListofVals. A corresponding FRL function is FGET. For example access.Attr_val([id, est]:Task, [Identifier, EarlistStartTime]) is an example of an usage of this predicate, and this is used to determine the task identifier Identifier and the earliest start time of the task EST.

Assigning attribute values:
assign_Attr.val(AttrbList::Object Attribs, ValueList, NewObjectAttribs)

assigns new values to attributes of the given object and returns a list of new object attributes. The corresponding FRL function is FPUT. This is the standard method for changing the attribute of some object, for example assign_Fld.val( [5,1]::Task_F, [est,priority], NewTask_F) assigns values 5 and 1 to the earliest start time and priority attributes of the task whose attributes are given by the list Task_F and the new list of task attributes is denoted by NewTask_F. The order of the attributes in the object is maintained as can be seen from the predicate implementing assign_Attr.val. Objects are maintained in the system as a collection of attributes, for example a task in a jobshop scheduler can be represented as task([id::brake_job, est::5, let::15]), this object is often converted internally to the compound term form task(brake_job, 5, 15) where names of attributes are maintained implicitly by the position of the attribute values. It is easier to manipulate the object in the second representation especially during sorting and insertion operations during the course of the schedule.

Adding a value to an attribute:

add_Attr.val(OldAttribList::Object Attrib, NewVals, NewObjectAttribs) adds a value to one of the existing attributes of an object. If we have an object session with one of the fields enrollee, and the enrollment is currently being calculated then another value of the attribute can be added by a call to the predicate add_Attr.val/3 i.e. add_Attr.val([enrollee]::Task_F, [henry], New_Task_F) adds henry as a member of the enrollee attribute of Task_F.

Adding a new attribute:

add_Attrbs([order,new_attrib]::Task_F,[1,5],NewTask_F) adds new attribute values order and new_attrib to the orginial task attributes Task_F to give the new task attributes NewTask_F. This is the method by which an object can be dynamically expanded to accommodate new properties. The values assigned to these attributes are 1 and 5 respectively. The most useful purpose for this predicate is to create dynamic variables which will be used to determine the course of the schedule, for example if the user develops a new heuristic which allocates first resources
:- op(640, yfx, ':-').

%------ making a new object ------
make_object(ObjName, Attribs, Object) :-
    initialize_attribs(Attribs, InitializedAttribs),
    Object =.. [ObjName[InitializedAttribs]].

%------ initializing attribs ------
    initialize_attribs([], []).
    initialize_attribs([(Attrib::Val)|RestAttribs],[((Attrib::Val)|RestInitialize):-
    !],
    initialize_attribs(RestAttribs, RestInitialize).
    initialize_attribs([Attrib|RestAttribs],[(Attrib::[])|RestInitialize]) :-
    initialize_attribs(RestAttribs, RestInitialize).

%------ Accessing a Attr Value ------
access_Attr_val([], Object_F, []).
access_Attr_val([Attrib|RestAttribs]:Object_F,[AttribVal1|RestAttrVals]) :-
    access_val(Attrib::AttribVal1,Object_F),
    access_Attr_val(RestAttribs::Object_F,RestAttrVals),
    !.

access_val(Attrib::Value, List) :-
    member(Attrib::Value, List),
    !.

%------ Assigning a Attr Value ------
assign_Attr_val([], Obj_Attribs, []).
assign_Attr_val([Attrib1|RestAttribs]:Object_F,[NewVal|RestNewVals], N_Object_F):-
    select_attrib(Attrib1::OldValue, Object_F, Preceding_F, Following_F),
    append(Preceding_F,[Attrib1::NewVal, List1),
    append(List1, Following_F, N_Object_F),
    assign_Attr_val(RestAttribs::N_Object_F1, RestNewVals, N_Object_F).

%------ Appending a Attr Value ------
add_Attr_val([], Object_F, []).
add_Attr_val([Attrib1|RestAttribs]:Object_F,[AddedVal|RestAddedVals], N_Object_F):-
    select_attrib(Attrib1::OldValue, Object_F, Preceding_F, Following_F),
    append(Preceding_F,[[Attrib1::AddedVal|OldValue)], List1),
    append(List1, Following_F, N_Object_F1),
    add_Attr_val(RestAttribs::N_Object_F1, RestAddedVals, N_Object_F),
    !.

Figure 3-18: Prolog Predicates for Accessing an Object System
Figure 3-19: Prolog Predicates for Accessing an Object System Continued

which are currently assigned for the least amount of time, then a new attribute
current_allocation_time is created based on the current allocation time.

Deleting attributes:

dele_Atrrs(AttribList, AttrToBeDeleted, NewAttribList) deletes attributes
from AttribList as indicated in AttrToBeDeleted. NewAttribList is the new
attribute list. The corresponding FRL function is FREMOVE. This predicate
delete_Attrrs/3 performs the inverse operation of the predicate add_Atrrib/3.
Some attributes have to be dynamically purged from the object after the current
schedule cycle is over, for example current_allocation_time which is dependent
on the current resource allocation heuristic.

Finding objects:

find_object(Attribs::AttribVals, ObjectList, Object) finds an object with
a certain attribute value out of a given ObjectList. The predicate is used for
extracting objects from a given collection of objects, for example if we want to
find a tasks with taskid tsk1 and earliest start time 5, from a collection of tasks
then `find_object([taskid, est]::[tsk1, 5], TaskList, Task)` will return the task with the required attributes. `find_object/3` fails if Object with specified attributes does not exist in the given `ObjectList`.

### 3.7 Related Generic Software Environments

One of the main aims of this thesis is to re-use the ideas presented and develop a generic architecture for developing schedulers in different domains. In this section we discuss about design of generic architectures and present one generic system “KASE” [BN92] which has been also used for designing a transportation scheduler [Smi90].

#### 3.7.1 Software Abstraction

Synthesizing software systems by reusing previously developed components has long been a subject of considerable interest in software engineering. One of the most effective principles that has emerged for re-using software is abstraction. Abstraction consists of extracting the inherent, essential aspects of an artifact, while hiding its irrelevant or incidental properties. One of the ways in which abstraction fosters reuse is by providing a class of artifacts that can be instantiated or customized to produce several different artifact instances meeting different requirements. Procedural and data abstraction, encapsulation or information hiding, and parameterized programming are examples of some of the notable application of the abstraction principle in software systems.

The abstraction principle has also been used as the basis for automating the construction of artifacts that would normally require a creative process. For example Emycin [Mel80], an expert system shell was developed by abstracting the control structure of Mycin; abstracting out the process of building blackboard systems yielded `AGE` [NA79]. Commercially available expert system shells and application generators are based on different mixtures of design and process abstraction. More recently, abstraction has been successfully used in algorithm synthesis, e.g. the KIDS system [Smi90] contains abstractions of several different classes of algorithms in the form of algorithm theories which can be semi-automatically instantiated to synthesize specialized algorithms for several different problem instances.
In the KASE (Knowledge Assisted Software Engineering) project, Bhansali and Nii [BN92], [Gui90] have investigated the utility of abstracting software system designs and the design process. Designing software system is a creative and ill-understood process. Software systems are created by a small group of designers; however the process is rarely documented and the final design is typically not well documented. Consequently, it is difficult to understand and maintain the system, which in turn leads to poor reuse. Their approach to this problem consists of

1. Identifying useful classes of software systems and problems they solve.

2. Abstracting the design of the generic architecture for that class of problems.

3. Constructing specific systems semi-automatically by customizing the generic architecture using domain specific information.

Such an approach allowed them to reuse the architecture for multiple applications within the class, capture the process of software design which could be used to maintain the system [Bha92]. A guiding theme of the research was to provide a set of software tools that support the way humans design. Typically KASE provides design alternatives and default suggestions for architectural parameters, explanations for its suggestions, dependency maintenance between different decisions, and consistency checking. Based on this approach several domain specific applications can be constructed. The framework has been used to design two different systems belonging to the same problem-class by reusing a single generic architecture.

3.7.2 Framework for Domain Specific Software Design

Figure 3-20 shows an overview of the KASE approach. The shadowed boxes represent knowledge components that are part of KASE. A designer initiates the design process by first selecting a generic architecture from a library based on the problem class for his particular problem and the desired solution features. Associated with the generic architecture is a meta-model which establishes the vocabulary for describing the problem class. Also associated with the generic architecture is customization knowledge which contains knowledge for customizing the generic architecture and is
the basis of KASE's intelligent support. Kase also has a constraint checker that is used to check for the consistency of the design.

For example KIDS (Kestrel Interactive Development System) has been used to derive extremely fast and accurate transportation schedulers from formal specification. The speed of this scheduler derives from the synthesis of strong problem-specific constraint checking and constraint propagation code. KIDS is a program transformation system - one applies a sequence of consistency preserving transfor-
mations to an initial specification and achieves a correct and hopefully efficient program [Smi90]. The system emphasizes the application of complex high level transformations that help perform significant and meaningful decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context". We hope that decisions at that level will be both intuitive to the user and be high-level enough that useful programs can be derived within reasonable number of steps. The user typically goes through the following steps in using KIDS for program development:

_Developing a domain theory:_ An application domain is modeled by a domain theory (a collection of types, operations, laws and inference rules). The domain theory specifies the concepts, operations and relationships that characterize the application and supports reasoning about the domain via a deductive inference system. KIDS has a theory development component that supports the automated derivation of various kinds of laws.

_Create a specification:_ The user enters a problem specification stated in terms of the underlying domain theory.

_Apply a design tactic:_ The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing specification to a library routine), divide-and-conquer, backtrack, branch-and-bound and local search (hillclimbing algorithms) [Low91].

_Apply Optimizations:_ The KIDS system allows the application of optimization techniques such as expression simplification, partial evaluation, finite differencing, case analysis and other transformations [Smi90]. The user selects an optimization method from the menu and applies it by pointing at a program expression.

_Apply data type refinements:_ The user can select implementations for high-level data types in the program. Data type refinement rule carry out the details of constructing the implementation [BG91]

_Compile:_ The resulting code is compiled to executable form. In a sense, KIDS can be regarded as a front-end to a conventional compiler.
The user is free to apply any subset of the KIDS operation in any order - the above sequence is typical of our experiments in algorithmic design.

Our generic architecture also allows for developing a domain theory, creating a specification, apply a design tactic and developing customizable schedulers. The strengths of the generic scheduler environment is that it is specialized for a scheduling environment and we believe that it is easier to write a scheduling application under the framework as opposed to a more general purpose framework presented in KASE.

3.8 Summary
In this chapter the design of a scheduling framework is presented. The original question which we tried to answer was "Can a generic scheduler be designed that would significantly shorten the development time of individual schedulers?" and the response to it is the development of a generic scheduling framework which helps scheduler developers to develop customized schedulers. Any scheduling, as we have mentioned earlier is - "the assignment of resources to tasks subject to constraints" and in this manner all schedulers are really the same when it comes to the philosophy of scheduling. The difference comes about in two areas. The first one is the nature of tasks, resources and constraints in the system and the second one is domain specific information and strategy which may be very useful in getting efficient schedules. Nevertheless it is quite possible to systematically develop schedulers, as schedulers are closely related.

In order to develop the generic scheduler environment, the scheduling clusters concept was developed. A cluster is a group of scheduling problems which are closely related, for example examination scheduling problems, classroom timetabling problems, conference scheduling problems are a group of very closely related problems and belong to the same cluster called the timetabling scheduling cluster. Problems like truck dispatch schedulers or tanker dispatch schedulers belong to one area of related schedulers collectively developed under the transportation scheduling cluster. These two group of problems are also related to each other because both involve allocation of resources to tasks, and can be solved by a single algorithm which can
be specialized to both scheduling instances. The general algorithm is presented in Figure 3-11 while the specialized version of the same algorithm is presented in each individual scheduling cluster chapters. For example Figure 6-5 is a specialized instance of the algorithm for timetabling schedulers, while Figure 5-3 presents a specialized case of the same algorithm for transportation schedulers. In the current chapter an object oriented representation built in Prolog was presented to represent all the schedulers, and temporal constraints were handled by representing time as intervals and adapting Allen's relation for finding temporal intersections and updating time intervals.

The rest of the thesis in Chapter 4, 5, 6 and 7 develops different scheduling clusters and shows how the ideas developed in this thesis could be used to build practical scheduling applications.
Chapter 4

Jobshop Schedulers

"Better stop short than fill to the brim. Oversharpen the blade, and the edge will soon blunt. Amass a store of gold and jade, and no one can protect it. Claim wealth and titles, and disaster will follow. Retire when the job is done. This is the way of heaven."
- Tao Te Ching.

4.1 Introduction
Scheduling is a complex process involving several jobs, resources and constraints. In this chapter we describe the formulation of a cluster which we call the jobshop scheduling cluster. After a brief description of the jobshop cluster, we describe one instance of the jobshop cluster which is called the carshop scheduling problem. The ideas and algorithm used in the jobshop scheduling cluster provide a method to handle manufacturing scheduling type problems, as well as methods which can be re-used for other scheduling clusters described in Chapters 5, 6 and 7. Jobshop scheduling is a classical scheduling problem and probably studied the most in the operations research literature. We present a definition of the general jobshop scheduling problem, this definition defines a structure which fits many scheduling problems arising in business, computing, and other industrial problems, hence we name the entire cluster as the jobshop scheduling cluster. The jobshop scheduling cluster solves a
very common kind of scheduling problem - there are \( n \) jobs \( \{J_1, J_2, \ldots, J_n\} \) which have to be processed/scheduled through \( m \) machines \( \{M_1, M_2, \ldots, M_m\} \). The machines have alternatively been referred to as processors if the scheduling is done in the context of computer jobs, or they could be referred to as operators in carshop scheduling presented in more detail in this chapter. In general we can refer to them as resources.

Technological constraints, which are constraints imposed by the current machine setup (location, capacity), demand that each job should be processed through the machines in a particular order. For general jobshop problems there is no restriction on the form of technological constraints, each job has its own processing order and this may bear no relation to the processing order of any other job. If all jobs have the same processing order then it gives rise to a special class of problems called the flowshop problem. We motivate our discussion about the current cluster by presenting some examples of the various scheduling problems which fall into the category of jobshop scheduling problems.

Industrial Examples: Any manufacturing firm not engaged in mass production of a single item needs to schedule individual items. Each product will have its own route through the various work areas and machines in the factory. In the steel industry each size of rod or girder passes through a set of rollers in its own particular order and with its own particular temperature and pressure. In the printing industry the various activities to be scheduled are typesetting, printing, binding, and packaging, giving rise to a four machine flow-shop problem. Objectives in these problems could be to maintain equal amount of activities in all departments so that idle time of expensive skills and machines could be minimized, or could be to minimize cost of production.

Aircraft queuing up to land: This is an \( n \) job, one machine problem. The aircraft is a job and the runway is a machine. Each aircraft has a ready time, namely the earliest time it can get to the airport’s air-space and be ready to land. The objective may be to minimize the average waiting time of the aircraft before they land.
Space Station Scheduling: This interesting application has been studied by Kurtzman in [Kur88]. The detailed inspection of a prototypical space station plan would reveal a set of \( n \) jobs typically in the order of hundreds to be performed by \( m \) crew members typically 8 or less over a period ranging from several hours to several months. Each of the job has a default time for completion, although in some cases different crew members may have different skill levels and take different times to complete a job. The objective is to find a feasible schedule for all the onboard crew activities.

Other Scheduling Problems: Some other problems which fit into the jobshop scheduling cluster are scheduling of different programs on a computer, the processing of different batches of crude oil at a refinery, the repairs of cars in a garage, the manufacture of paints of different colors.

### 4.2 Carshop Scheduling Problem

The carshop scheduling problem is an instance of the jobshop scheduling cluster and involves scheduling repair jobs on cars, given restrictions on operator availability and other resource/time constraints. The problem is solved by using the intelligent generate and test scheduling approach developed in this thesis. The primary objective of the schedule generator is to generate a complete schedule, i.e. successfully schedule all jobs within their constraints, while the user has the option to choose secondary objectives like minimizing total cost of allocation, or minimizing maximum completion time of jobs. The system is built on a user-extensible knowledge base of rules and heuristics written in Prolog. The emphasis in the system is on providing a flexible AI problem representation and also collecting some empirical results on the performance of different heuristics in the system. The basic scheduler design has also appeared in [SF92].

Carshop scheduling like other scheduling problems is an NP-Complete problem. The chief dimensions of any scheduling problem are \textit{Tasks}, \textit{Resources} and \textit{Time} which map in the carshop scheduling instance into \textit{car repair jobs}, \textit{operators with tools/machines} and \textit{time units}. Each car repair job may be comprised of several
tasks to be done on the same car, and is considered complete when all the tasks in
it are finished. A related problem has been solved in [DSH88], where the emphasis
is on sequencing.

The current scheduler is built using an intelligent generate and test scheduling
strategy, where generate and test is carried out in stages; care is taken to use heuris-
tics to minimize the amount of backtracking and get a good solution. Reported
methods for scheduling range from “constraint directed” scheduling [Fox87] which
focuses on knowledge representation and integration of constraints to the search pro-
cess during scheduling; to an Expert Systems approach for scheduling as described
in [GKM90, DS85, SC87]. Each schedule can have multiple objectives some of which
can be conflicting. Some methods for handling conflicting objectives are presented
in [Ber90]. In the present scheduling problem the primary objective of the schedule
is to schedule all the jobs, subject to resource allocation and temporal constraints.
Some of the secondary objectives considered are minimizing cost of schedule, where
costs are incurred because operators have a cost associated for every unit period of
allocation time; another secondary objective could be minimizing total completion
time(makespan) of the schedule.

As the solution is of a generate and test nature, the scheduler picks up one of
the available combination of task and operator selection heuristics, and generates
complete schedules. The best solution in a finite number of iterations is reported.
More details about the scheduling process follow in the chapter. Finding an optimal
schedule with respect to the secondary criterion is not guaranteed as it may take
exponential time, the system instead tries to report the best solution it obtains in a
certain number of iterations.

4.3 Carshop Scheduling Components
We briefly describe all the input, output, tasks, resources and constraints, objectives
of the carshop scheduler which is the instance of Jobshop scheduling cluster. The
task in the carshop scheduling problem is to schedule operators to cars needing
repair. The carshop takes orders in the morning and the entire schedule for the day
is generated in a batch mode operation.
4.3.1 Scheduler Input

The scheduler is required to schedule a set of jobs $J = \{J_1, \ldots, J_{n_J}\}$, given a set of physical resources $R = \{R_1, \ldots, R_{n_R}\}$ and a set of constraints $C = \{C_1, C_2, \ldots, C_{n_C}\}$. Here $n_J$, $n_R$ and $n_C$ are the number of jobs, resources and constraints respectively. Each job $J_i$ contains a set of tasks; $J_i = \{T^i_1, \ldots, T^i_{n^t_i}\}$, where $n^t_i$ is the number of tasks in job $J_i$.

4.3.2 Scheduler Output

The scheduler needs to find values for variables which are defined below:

**Task Start Time:** The scheduler has to instantiate the starting time of each of the tasks $St^i_k$ ($1 \leq i \leq n_J, 1 \leq k \leq n^t_i$).

**Resource Allocation:** The scheduler has to allocate for each of the tasks $T^i_k$ of a job $J_i$, a resource set $R^i_{kt}$ ($1 \leq i \leq n_J, 1 \leq k \leq n^t_i, 1 \leq t \leq END\_TIME$), which indicates the set of resources assigned to task $k$ of job $i$ starting at time $t$, given that the entire schedule lasts between the time 1 and END\_TIME.

The time variable in the schedule is measured in terms of time slots, this assumes integral values ranging between 1 and END\_TIME, which is the last time slot in the scheduling operation of a day.

4.3.3 Schedule Data

The input to carshop scheduling problems consists of static and dynamic information. The static information provides data about the resource requirements of each task, the performance characteristics of resources in terms of time taken by different operators to finish different tasks, and the starting weights associated with each task required by different heuristics used in the system. The dynamic data consists of task input information in terms of a list of cars to be scheduled along with tasks to be performed in each of them, and information about resource availability in terms of availability periods of different operators. The constraints in the system can be dynamically stated with the problem, or stored as predicates in the form of static data. Some sample carshop information is provided in Table 4.1.
Jobs/Tasks:

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>Priority</th>
<th>ArvTime</th>
<th>Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix transmission</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>tuneup</td>
<td>2</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>car2</td>
<td>fix brakes</td>
<td>2</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>fix transmission</td>
<td>1</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>car3</td>
<td>fix gasket</td>
<td>3</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4.1: Sample Task/Resources in a car shop Scheduler

<table>
<thead>
<tr>
<th>Operator</th>
<th>Avail Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>al</td>
<td>(4,7),(9,11)</td>
</tr>
<tr>
<td>bert</td>
<td>(1,9),(11,16)</td>
</tr>
<tr>
<td>chip</td>
<td>(2,7),(9,14)</td>
</tr>
</tbody>
</table>

Resources:

<table>
<thead>
<tr>
<th>Op</th>
<th>Cost/Time</th>
<th>Time Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>brakes</td>
<td>gasket</td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>bert</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Heuristic1 Wt</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Heuristic2 Wt</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Sample Operator Performance Data

Jobs:

A list of cars with problems is provided as shown in Table 4.1. Each car is considered as a job while different tasks of a particular car make up the job. Associated with each task in a car is a priority added by the scheduler user, tasks with greater priority indicate a preference of the task to be scheduled before other tasks with a lower priority. The heuristic weights in conjunction with priorities determine the order of dispatch of the task during scheduling. Each job has an arrival time, which indicates the earliest possible time after which it can be scheduled. Due dates for all jobs are provided or are defaulted to the END_TIME of the schedule period.

Resources:

The resources in the system are operators with a range of availability hours. Each operator is available only for certain time slots during the day. The availability time for each operator is denoted by a set of time intervals, as shown in Table 4.1 and each interval has a lower and upper bound. As the operator gets assigned to different tasks, the availability hours are suitably updated, depending upon the task assignment.
Resource Requirements for Tasks:

Each task requires resources in terms of operators assigned for a length of time. The resource requirements for specific jobs do not change very often, i.e. the time requirement and operator requirement for a task like brake job remains the same. Each operator incurs a certain cost for each time slot he is allocated to the task, as indicated by the second column of Table 4.2. The times taken by each operator to perform tasks are also shown in the different rows of Table 4.2. For example assigning a1 to a task will incur a cost of 4 cost units for each time unit he is assigned, and a1 can finish a brake job in 1 unit of time. Also a1 is available for assignment only in the time intervals (4,7) and (9,11).

Priorities:

The priority of tasks indicate a preference ordering while dispatching them to be scheduled. Priorities are be considered as soft constraints on the order of task dispatch which can be overridden, these are integers in the range of 1 to 5, with the default priority being 1.

4.3.4 Scheduler Constraints

The scheduler searches for solutions trying to instantiate $S_{ik}$ and $R_{kt}$ ($1 \leq i \leq n_J$, $1 \leq k \leq n_T, 1 \leq t \leq END\_TIME$) in an environment involving numerous constraints. The constraints are categorized into Temporal Constraints and Resource Constraints.

Temporal Constraints

Start/End Time: Absolute times are converted into time represented in terms of slots. The schedule starts at time slot 1, and ends at time slot END\_TIME. All jobs in the schedule workshop must be started and completed between these time slots, furthermore a task must be completed before its due time if provided as input. If no due date is provided then it is assumed to be the END\_TIME of the schedule.
**Precedence Relations:** These occur in the list of constraints given as input to the system. A precedence relation states that the preceding task must be finished before the following task is scheduled. During the scheduling process the preceding tasks are scheduled before the tasks which need to follow. The precedence relationships hold only between two tasks of a job, and there are no explicit precedence relationships between jobs. Precedence relationships in Prolog are expressed by the predicate `precedes/3` where the second argument is the preceding task followed by a list of all possible tasks it must precede. Some sample precedence relationships are provided below:

```
precedes(car1, oilchange, [tuneup]).
precedes(car2, brakejob, [tuneup, brakeFluidChange]).
```

**Forbidden Resource Allocation:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Forbidden Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>leon</td>
<td>fix_fender</td>
</tr>
<tr>
<td>albert</td>
<td>brakejob</td>
</tr>
</tbody>
</table>

**Infeasible Op Combinations:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>albert</td>
<td>leon</td>
</tr>
<tr>
<td>joe</td>
<td>bert</td>
</tr>
</tbody>
</table>

Table 4.3: Sample Resource Constraints for a carshop Scheduler

**Resource Constraints**

Resource constraints curtail the allocation of resource combinations to tasks. Some resource constraints are shown in Table 4.3.

*Constraints forbidding an operator from working on certain tasks:* The management reserves the right to preclude the assignment of an operator from a set of tasks. The decision is mainly based on the competence of an operator.

*Constraints forbidding certain operators from working together:* There may be a set of operators who cannot work together on a single task. One possible reason for such a constraint could be that the set of operators do not make a good team.
4.3.5 Scheduling Objectives

The problem has multiple objectives, the primary objective is to schedule all the jobs within temporal and operator availability constraints such that the schedule completes within the prescribed time. The secondary objective may be any of the following:

- Minimize the maximum completion time
- Minimize the cost of allocation.

The objective for obtaining any schedule \( S \) can also be restated as an Operations Research Formulation.

Solve:

\[
\begin{align*}
\mathcal{R} | \text{nopmnt, } & C, r_i | \min(\max_{i \in S} Cst(J_i)) & \text{subject to } \max_{i \in S} Td(J_i) = 0 \\
\text{or} & \\
\mathcal{R} | \text{nopmnt, } & C, r_i | \min(\sum_{i \in S} Cst(J_i)) & \text{subject to } \max_{i \in S} Td(J_i) = 0
\end{align*}
\]

where

- \( S \) is the schedule which is an ordered set,
- \( \mathcal{R} \) is the resource set,
- \( C \) is the constraint set,
- nopmnt is no pre-emption,
- \( r_i \) is any time of arrival of job \( J_i \),
- \( Ct(J_i) \) is the completion time of the job \( J_i \),
- \( Cst(J_i) \) is the cost of completing job \( J_i \) and is calculated as
  \[
  Cst(J_i) = \sum_{j=1}^{n_i} Cst(T_{ij}^i),
  \]
- \( Td(J_i) \) is the tardiness of job \( J_i \) which is the difference between the completion time and due date of a job, the secondary objective is minimization of \( \max(\text{CompletionTime}) \)
  or minimization of total cost of job allocation.
4.3.6 Choice of Scheduling Measures
Each schedule has a measure based on which the quality of the schedule is judged, and schedules which may be good with respect to one measure may not be good with respect to another measure. The generated schedules are of two types:

**Complete Schedule:** A schedule is said to be **complete** if it successfully schedules all jobs within their constraints.

**Feasible Schedule:** A schedule is said to be **feasible** even if it does not schedule all jobs, as long as it does not violate any constraints.

Failure to find a complete schedule causes the carshop scheduler to find a feasible solution and stop. However if a number of complete solutions are available then one solution which is the best amongst these, with respect to a given measure is reported. Heuristics as explained later are used in order to prune the search space.

4.4 Scheduler Components
The jobshop scheduler system can be functionally decomposed as shown in Figure 4-1. The jobshop scheduling problem is decomposed into tasks, resources and constraints; these are fed into the Schedule Pre-analyser, which checks the partial feasibility of the input. The core scheduling engine consists of a task dispatcher, resource allocator and constraint propagator which generate the schedule. The control is provided by the heuristics selector, which guides the schedule by selecting the the task selection and operator selection heuristics.

4.4.1 Task Dispatcher
This determines the order in which different tasks are dispatched in the system. Tasks are **first ordered** purely based on task dispatch heuristics. The task which is selected for dispatch satisfies all temporal constraints. More details follow in the section on schedule iteration.
4.4.2 Resource Allocator

The resource allocator allocates resources to a schedulable task. Since a number of operator-task allocation combination exist for each task, the resource allocator is guided by resource selection heuristics to determine operator selection for each task. The resource allocator also determines the start time of each task, as it depends both on the temporal constraints on the task, and the periods of availability of the operator.
4.4.3 Constraint Propagator

Once a task is allocated a resource and has known start and completion time, the constraint propagator determines the earliest starting times of other tasks in the same job. This is possible now based on the partial ordering between tasks of the same job and also the fact that the exact completion task of one task has been exactly determined in the current iteration. Since only one task of a job is doable at any given instance of time, the earliest start time of all tasks of the same job are pushed to at least the end time of the scheduled task. The starting time of some tasks which were previously uninstantiated because of precedence constraints, can also now be instantiated if the preceding task has been scheduled.

More sophisticated handling of temporal constraints can be found in [Rit86] and [Smi83], both of which provide a framework for propagating temporal constraints in a broader scheduling context.

4.4.4 Heuristics Selector

The heuristics selector provides flow of control information to the Task Allocator and Resource Dispatcher units. Each task dispatch heuristic associates a set of weights with all the tasks which need to be scheduled. All tasks are ordered in decreasing order of heuristic weights, to form an ordered set OrderedTasks. resource selection heuristics provide control information to the Resource allocator, and at each iteration step, resources are arranged in a resource list in decreasing order of preference i.e. with the most preferred resource first. The first permissible resource is allocated to the selected task.

4.5 Grammar for Jobshop Schedulers

We describe grammar for recognizing task, resource, constraints and expressing schedule heuristics. Prolog Definite Clause Grammar rules are used to parse the input to the scheduling problem. The top level rule is scheduler which collects all the required information about the current scheduler. The information fields which define the scheduling cluster are, dataset information - which keeps track of
the problem which is being solved, the tasks in the system, resources in the system, constraints in the system, costs of resource allocation to tasks. Other information collected include schedule parameters to be used in the current schedule - this includes the maximum number of iterations which need to be made for each heuristics pair. phrase/2 is the Prolog predicate which parses the input stream and collects scheduler specific information.

```
scheduler(Dataset, Tasks, Resources, Constraints, Heuristics, Costs, SchedParams, Objective)  
    → dataset(Dataset),
        tasks(Tasks),
        resources(Resources),
        constraints(Constraints),
        schedule_heuristics(Heuristics),
        costs(Costs),
        schedule_parameters(SchedParams),
        objective(Objective).

dataset(Dataset)  → [dataset, ':'], Dataset, ['.'].

tasks(Tasks) → [tasks, ':'],
                   tasks1(Tasks),
                   delimiter.

tasks1(Tasks) → task(Task),
                   tasks1(RestTasks),
                   [Tasks=[Task|RestTasks]].

tasks1([]) → [].

resources(Resources) → [resources, ':'],
                        resources1(Resources),
                        delimiter.

resources1(Resources) → resource(Rcs),
                        resources1(RestRcs),
                        [Resources=[Rcs|RestRcs]].

resources1([]) → [].
```
4.5.1 Grammar for Task Recognition

Task fields in the case of a carshop scheduler consist of a task identifier, a task name which could be for example be a fix brakes task, or a fix lights task. Other task fields include the earliest start time of the task which may be left unspecified and priority of the task.

\[
\text{task}(\text{task}([\text{task\_id}::\text{TaskId}, \text{task\_name}::\text{Name}, \text{est}::\text{EST}, \text{priority}::\text{Priority}[\text{ExtraFields}]])) \\
\rightarrow \text{taskId}(\text{TaskId}), [','] \\
\rightarrow \text{task\_name}(\text{Name}), [','] \\
\rightarrow \text{est}(\text{EST}), [','] \\
\rightarrow \text{priority}(\text{Priority}), [','] \\
\rightarrow ['#,'] \\
\rightarrow \text{extra\_fields}(\text{ExtraFields}) \\
\rightarrow ['#,'] \\
\text{taskId}(\text{TaskId}) \rightarrow ['\text{task\_id}', ':', \text{TaskId}], \\
\rightarrow \{\text{atomic}(\text{TaskId})\}. \\
\text{task\_name}(\text{Name}) \rightarrow ['\text{task\_name}', ':', \text{Name}], \\
\text{est}(\text{EST}) \rightarrow ['\text{est}', ':', \text{EST}], \\
\text{priority}(\text{Priority}) \rightarrow ['\text{priority}', ':', \text{Priority}].
\]

In the current example shown in Table 4.4 there are two jobs to be performed on cars 1 and 2. Each of these cars have two tasks each, for example car 1 has task fix transm and task tune up. The earliest start time of all the tasks are 0 and the priority varies from 1 to 4.

4.5.2 Grammar for Recognizing Resources

The resource field for the carshop schedulers consists of the resource identifier, the current load on the resource which is an indication of the amount of time the resource is already allocated. The other fields include the skill level of the resource, the number of hours the current resource is available for allocation and the actual window for which it is available. Additional extra fields can be added depending
on specific members of the cluster. In other instances of the jobshop schedulers the skill level field can be used as an index of efficiency, and we can assume that a higher skill level means higher efficiency.

resource(rcs([rcs_id::RcsId,load::Load,skill_level::Skill_Lev,
               av_hours::AvHours,av_window::AvWindow|ExtraFields]))
    → rcs_id(RcsId),'|',
        load(Load),'|',
        skill_level(Skill_Lev),'|',
        avail_hours(AvHours),'|',

Table 4.4: Sample Carshop Task Translation
avail_window(AvWindow),[';'],
[',#'],
extra_fields(ExtraFields)
[',#].

rcs_id(RcsId) → ['rcs_id',':'],
[RcsId].

skill_level(Skill_Level) → ['skill_level',':'],
[Skill_Level],
{integer(Skill_Level)}.  

load(Load) → ['load',':'],
[Load],
{integer(Load)}.  

avail_hours(Av_Hours) → ['av_hours',':'],
[Av_Hours].

avail_window(AvWindow)
→ ['av_window',':'],
[''],
window(Window),
avail_window1(RestWindows),
[''],
{append(Window,RestWindows,AvWindow)}.  

window([av(W1,W2)]) → ['(',W1,',',W2,')].

In Table 4.5 sample input resources and their translation to the intermediate Prolog form has been shown. The first resource has resource identifier a1, and is not allocated to any task as shown by load 0. His skill level is 5 and is available for only 5 time units of work. The available work hours are provided as a value to the field av_window and the value of available hours is [(4,7),(9,11)]. The second resource bert has a pre-assigned load of 0 time units, has a skill level of 3 and is available for 8 time units in the interval [(1,5),(8,12)].
4.5.3 Grammar for Recognizing Constraints

The three types of constraints recognized in this scheduling cluster are task, resource and temporal constraints. The task constraints specify that some tasks should not have some resources allocated to them, the resource constraints specify that some resources should not be allocated to certain tasks. Explicit temporal constraints in this case help in specifying the precedence relationship between different tasks, they indicate which tasks should precede other tasks.

```
constraints(Constraints)
    → [constraints,'::'],
        task_constraints(Task.Cons),
        resource_constraints(Rcs.Cons),
```
temporal_constraints(Temp_Const),
{ append(Task_Const,Rcs_Const,Cons),
append(Cons,Temp_Const,Constraints)
}.

constraints([]) → [].
task_constraints(Task_Constraints)
→ ['task_constraints',',',],
forbidden_task_rcs_list(Task_Constraints).
task_constraints([]) → [].

rcs_constraints(Rcs_Constraints)
→ ['rcs_constraints',',',],
forbidden_rcs_task_list(Rcs_Constraints).

rcs_constraints([]) → [].
temporal_constraints(Prec_Constraints)
→ prec_constraints(Prec_Constraints).

prec_constraints(Prec_Constraints)
→ ['precedes',',',],
[TaskId],
[Task,','],
preceding_tasks(Tasks),
[']',',',],
prec_constraints(RestPrecs),
{Prec_Constraints=[precedes(TaskId,Task,Tasks)]RestPrecs}.

prec_constraints([])
→ [].

preceding_tasks(Tasks)
→ [Task,','],
preceding_tasks(Tasks1),
{Tasks=[Task|Tasks1]}.

preceding_tasks([Task])
→ [Task].
schedule_heuristics ::
  Task.Heuristics : pre_defined;
  Heuristic_Id : 1;
  Operation : wstf;

  Task.Heuristics : pre_defined;
  Heuristic_Id : 2;
  Operation : wltf;

  Task.Heuristics : user_defined;
  Heuristic_Id : 3;
  Operations :
    [ Operation : select;
      Conditions : [(standard,priority),>=,3];
      Operation : sort
      on Primary (user_defined,secondary_cost) asc;
    ];
  Task.Hs_Execution_Order: [2,1,3];

  task.ordering_hs(1,pre_defined,wstf).

  task.ordering_hs(2,pre_defined,wltf).

  task.ordering_hs(3,user_defined,[[select,
    [(standard,priority),>=,constant,3]],
    sort,(user_defined,secondary_cost,asc),[]]).

Table 4.6: Sample Constraint Translation for Carshop scheduler

An example of a simple precedence constraint in the case of the carshop scheduler can be precedes : car1 fix_transm [tune_up]; This constraint specifies that for car1 the task fix_transm should precede the task tune_up.

4.5.4 Expressing Carshop Scheduler Heuristics

Schedule heuristics guide the system in making good choices for task and resource ordering. There are three heuristic examples shown in Figure 4.6.

The first two heuristics are pre-defined in the system while the third one is an example of an user defined heuristic. The first heuristic is the weighted shortest task first heuristic which dispatches shorter task first during the schedule, the second heuristic is the weighted longest task first which dispatches the longest task first while scheduling. The third heuristic is more interesting, in this the user has asked the system to dispatch first all tasks whose priority is greater than equal to 3. All
tasks whose priority is less than 3 are not dispatched for schedule and furthermore
the tasks are sorted based on a new user defined field called secondary_cost, with
tasks having lower secondary costs being dispatched earlier. The order in which
heuristics will be applied in the system is [2, 1, 3].

4.6 Scheduling Heuristics
This section describes some of the heuristics used in the scheduling process. The
heuristics are used for selecting the next task to be scheduled, and selecting an
operator to be assigned to the selected task.

4.6.1 Heuristics for Task Selection
In order to get task orderings to be used in the serial dispatch heuristics, certain
measures for evaluation are required, the common measures used in this implemen-
tation are average time duration of a task and its priority.

The heuristic weights associated with each task selection heuristic are based
on the average time taken for the task to be completed. There are no heuristic
weights associated with operators who are ordered by operator selection heuristics
depending on the task to be performed or their present availability. Sample clauses
for associating heuristic weights with tasks are given below:

\%
heuristic_wts(Heuristic,[(Task1, Wt),..., (Taskn, Wt)]).
heuristic_wts(wstf,[(fix brakes, 5), (change muffler, 4), (replace gasket, 3), (tune up, 1)]).
heuristic_wts(wlts,[(fix brakes, 1), (change muffler, 2), (replace gasket, 3), (tune up, 5)]).

Dispatch weights are calculated as the product of heuristic weights and priority
of the task. During subsequent iterations the dispatch weights are increased by a
random factor.

The following are some heuristics used to obtain dispatch ordering amongst tasks:

**Highest priority first**

This heuristic dispatches tasks purely based on their priorities, tasks with higher
priorities are dispatched earlier.
Weighted shortest task first (wstf)

This heuristic uses the average time duration of a task as the heuristic parameter in order to obtain a rating. Tasks which can be completed in a shorter time are given higher weights.

Weighted longest task first (wltf)

The heuristic parameter used here is again the task duration. In this the task which takes the longest time is given preference over all other tasks and attempted to be scheduled earlier. Longer tasks are considered more difficult because they require resources for a longer time. This heuristic is used with the intention of trying to schedule the more difficult tasks first, and doing the easier tasks later.

4.6.2 Operator Selection Heuristics

Heuristics are required to allocate operators to perform the selected task. Some heuristics used to select operator orderings are:

Minimizing operator workload

This heuristic guides operator allocation by trying to maintain minimum operator workload amongst all operators. The number of time slots of work put in by each operator is maintained. All the operators are ordered in an increasing order with respect to their assigned hours i.e. operators with a lower number of assigned hours precede operators with a higher number of assigned hours. Ties between operators are resolved by picking up the operator with a lower index, from the set of ordered operators.

Schedule earliest finishing operator/operators first

This heuristic guides the search by choosing the available operator/operators who can finish the task in the shortest time. If there is a tie between two operator(s) or operator combinations then the operators with a lower workload are selected.
User preference allocation

The user preference ordering tries to allocate resources to tasks guided by user preference. A set of predicates user_preference(Tsk, OrderedOperatorList) are facts indicating the user preference of operator allocation to task Tsk. The first feasible operator assignment from the list OrderedOperatorList is done to the task Tsk. The schedule produced may not be good with respect to either time or makespan as allocation is not a function of the current work load of the operators.

4.7 Jobshop Scheduling Algorithm

The jobshop scheduling problem and the main functional units in the scheduler have been described in the previous sections. The generate and test scheduling engine is presented now in Figure 4-2.

In the current section a description of the scheduling process is provided based on the Prolog implementation of the system. The scheduling process starts by getting input from an X window interface system. The input is recognized by using a DCG grammar and the input is stored in a Prolog form in a file 'car_intermediate.pl' and is reconsulted during the second phase when actual schedules are generated. The main scheduling predicate is carshop_schedule which invokes other scheduling predicates and provides the user with schedules he/she has asked for. The initialization step in scheduling makes sure all scheduling counters are set to initial values, the tasks and resources are pre-ordered before the schedule is generated by invoking generate_schedule.

The predicate ordering.heuristic.execution orders the current set of tasks and resources to get a set of ordered tasks and resources. This ordering does not mean that tasks will be dispatched only in that order. The ordering is dynamic and as the scheduling proceeds, dispatch order of tasks vary depending on the constraint propagation done during each schedule cycle.

generate.schedule generates all the schedules and monitor.schedule monitors the goodness of each schedule and makes sure that schedule generation continues till all the required number of schedules have been produced. monitor.schedule implements a failure driven loop and terminates when the required number of scheduling
Input: Sets TaskList $T$, AvailableResources $R$, constraints $C$, earliest start time $EST_j^i$ of each task $T_j^i$, Heuristic Weights, Heuristics, MaxIterationsToTry, $N$, Initial Dispatch window $DW_1$ and $DW_2$.
Output: "Good" Feasible Schedule $S$.

S1: **Task Selection:**
1. If ($T$ is empty) goto S5. /* end of current schedule */
2. Select task $T_j^i \in T$ within current dispatch content window (DCW) i.e. ($DW_1 \leq EST_j^i \leq DW_2$).
   where $T_j^i$ has maximum heuristic dispatch weight in the DCW determined by task ordering heuristic.

S2: **Resource Selection:**
1. Select a resource $R_j^i \in R$ based on the resource selection heuristic subject to resource constraints.
2. If no resource found goto S6.

S3: **Allocation and Updating:**
1. Assign $R_j^i$ to $T_j^i$.
2. Calculate finish time $FT_j^i$; remove $T_j^i$ from $T$.
3. Calculate new Availability hours of resource $R_j^i$.
4. If ($R_j^i$.Available Hours == Nil) then remove resource $R_j^i$ from $R$.

S4: **Constraint Propagation:**
1. For all partially ordered tasks $T_k^i$ such that $precedes(T_j^i, T_k^i)$ set $EST_k^i = FT_j^i$.
2. Set $EST_k^i = FT_j^i$ for all tasks with known EST's. /* Tasks with unscheduled preceding tasks have uninstantiated EST's */
3. Update the availability of resource $R_j^i$ from step S3.
4. Calculate new dispatch window boundaries $DW_1$ and $DW_2$ and goto S1.

S5: **Calculations:**
1. Calculate cost of schedule.
2. If current schedule $S$ is cheaper than the previous best schedule then store $S$.
3. If (#ScheduleIterations $> N$) goto S9.

Figure 4-2: Generate and Test Algorithm for Jobshop Schedulers

iterations have been finished. Once each schedule has been generated, then the next schedule is computed by the predicate gen_next_schedule which permutes the given set of tasks to achieve a new task ordering and starts the scheduling process from
S6: **Preferential Dispatch Weight Adjustment:**
1. Preferentially increase heuristic weights of all current unscheduled tasks $T^i_j$ in $T$.
2. Preferentially increase heuristic weights of all $T^i_k$ where $precedes(T^i_k, T^i_j)$ and $T^i_j \in T$
3. goto S8.

S7: **Dispatch Weight Adjustment:**
1. Update heuristic dispatch weights of all $T^i_j \in$ original tasklist $T$ by multiplying original heuristic dispatch weight with a random factor.

S8: **Re-Initialization**
1. Reset tasklist $T$ with original tasks having updated weights.
2. Calculate initial dispatch window boundaries.
3. Increment #ScheduleIterations by 1.
4. goto S1.

S9: **Schedule Output:**
1. Print best solution $S$ generated.
2. Stop.

Figure 4-3: Generate and Test Algorithm for Jobshop Schedulers Continued

the start. The new task ordering is achieved by associating random weights with each task.

Each schedule iteration consists of a fixed number of steps. These steps are carried out until all the tasks are scheduled or the heuristics fail to provide a solution with the given pair of task and resource selection heuristics. The main features of a typical iteration are illustrated in Figure 4-4 by considering an example. Figure 4-4(a) indicates the current operator availability. The task selection heuristic decides the next task to be scheduled as $T^1_1$, and the operator selection heuristic determines op1 as the most suited operators for the selected task. Allocation and updation of resources are shown in the Figure 4-4(c). Once the allocation has been done and the partial schedule updated, a new set of tasks contend to be dispatched.

### 4.7.1 Task Selection

In each iteration tasks are chosen from the set of *earliest schedulable tasks*, the starting times of these tasks fall within the dispatch contention window shown by the shaded region in Figures 4-4(b),(d).
Dispatch Contention Window

At each stage of the scheduling iteration, certain tasks contend to be dispatched for scheduling. The task selection algorithm is applied to tasks lying within the dispatch contention window.

The size of the dispatch contention window is set before generating any schedule, it may vary from 0 to as large as the entire duration of the schedule, also the left boundary of a dispatch contention window is determined by the lowest EST’s among unscheduled tasks. Figure 4-5 illustrates the concept of dispatch contention
windows, a dispatch contention window of size 5 is used in the top half of the figure initially spanning slots 2 to 7. Tasks $T_1, \ldots, T_4$ whose earliest starting time are within the window compete for dispatch. The other tasks $T_5$ and $T_6$ have their earliest start times larger than the upper bound of the dispatch contention window, hence do not contend for dispatch.

On applying the task selection heuristics task $T_1$ is selected to be dispatched, and the dispatch contention window moves forward. Task $T_2$ has the next least EST, so the dispatch contention window starts from EST of $T_2=3$ to 8. New task
T5 comes into the dispatch contention window and now tasks T2,T3,T4,T5 contend for dispatch.

If a dispatch contention window of size 0 is used in the system, as shown in Figure 4-5(c) then the task with the least EST is dispatched. Dispatch contention may still need to be resolved between tasks having the same lowest EST's, i.e. between tasks T1 and T2 as shown in Figure 4-5(c).
choose.update_resources(Task,Rcs,AssignedRcs,UpdatedRcs) :-
    current_resource_list(Rcs,Heuristic),
    choose.update_resources(Rcs,Heuristic,Task,Rcs,AssignedRcs,UpdatedRcs).

% No hs ie. allocated without any particular order
choose.update_resources(user_defined,Task,Rcs,rcs(AllocRcs,F),UpdtRcs) :-
    select(rcs(Rcs,F),Rcs,RestRcs),
    determine.start_time(Task,rcs(Rcs,F),TimeSlotUsed,RestAvHours,EST,S,Time,F,Time),
    update.rcs.avail(EST,TimeSlotUsed,F,Time,RestAvHours,UpdtHours),
    time.calculation(Task,rcs(Rcs,F),UpLoad,UpRemTime),
    update.resources(user_defined,rcs(Rcs,F),UpLoad,UpdtHours,UpRemTime,RestRcs,UpdtRcs),

choose.update_resources(min_load,Task,Rcs,rcs(AllocRcs,F),UpdtRcs) :-
    select(rcs(Rcs,F),Rcs,RestRcs),
    determine.start_time(Task,rcs(Rcs,F),TimeSlotUsed,RestAvHours,EST,S,Time,F,Time),
    update.rcs.avail(EST,TimeSlotUsed,F,Time,RestAvHours,UpdtHours),
    time.calculation(Task,rcs(Rcs,F),UpLoad,UpRemTime),
    update.resources(min_load,rcs(Rcs,F),UpLoad,UpdtHours,UpRemTime,RestRcs,UpdtRcs),

choose.update_resources(e_time,Task,Rcs,rcs(AllocRcs,F),UpdtRcs) :-
    earliest.fin.rcs(Task,rcs(Rcs,F),RestRcs),
    determine.start_time(Task,rcs(Rcs,F),TimeSlotUsed,RestAvHours,EST,S,Time,F,Time),
    update.rcs.avail(EST,TimeSlotUsed,F,Time,RestAvHours,UpdtHours),
    time.calculation(Task,rcs(Rcs,F),UpLoad,UpRemTime),
    update.resources(e_time,rcs(Rcs,F),UpLoad,UpdtHours,UpRemTime,RestRcs,UpdtRcs),

Figure 4-7: Sample Prolog Predicates for Resource Selection/Update

The predicate earliest.schedulable.tasks/5 determines the set of earliest schedulable tasks. Since the tasks are already sorted increasing in time, the predicate easily determines the members of the earliest schedulable set by checking if earliest start times of a task is within the dispatch contention window. \( W_1 \) is the lower boundary of the dispatch contention window and all tasks for which \( EST \leq W_1 + DCWS \) are in the set of earliest schedulable tasks.

choose.task/3: This chooses the task determined as the most appropriate task by the task selection heuristic from the list of unscheduled tasks. The task is among one of the contending earliest schedulable tasks. This predicate first determines the boundaries of the dispatch contention window, then determines all tasks which can start within the dispatch contention window. Associated with each task is a dispatch weight which depends on the current task selection heuristic. The task chosen for dispatch in the current scheduling cycle is the one with the highest dispatch weight.
4.7.2 Choosing and Updating Operator Availability

`choose_update_resources/4` chooses the resource determined to be most appropriate by resource selection heuristics. Resources availability is also updated by the same predicate because it takes less time to do them together and also it is convenient as no extra information about resources needs to be passed around. This predicate is shown in the Figure 4-7. The first `choosing and updating resource` predicate selects a resource based on a user defined heuristics, first the resource is selected simply by the `select` predicate, then the starting time of the task is determined by `determine_start_time/7`. Since each resource takes a different time for completing a task, the finish time of the task can now be determined for the current resource and the availability of the resource is now updated based on the start and end time of the current task, updation of resource availability is shown in Figure 4-8.
propagate_cns(task(Task_F), alloc(Alloc_F), Tsks, TimeOrderedTasks) :-
  access_Fld_val([f_time]: Alloc_F, [F_Time]),
  precedence_prop(task(Task_F), F_Time, Tsks, UpTsks1),
  temporal_prop(task(Task_F), F_Time, UpTsks1, UpTsks2),
  order_tasks(UpTsks2, TimeOrderedTasks).

precedence_prop(task(Task_F), F_Time, Tasks, UpTasks) :-
  access_Fld_val([task_id, task_name]: Task_F, [TaskId,TaskName]),
  setof(precedes(TaskId, Task, SuccTsks), precedes(TaskId, Task, SuccTsks), Constraints),
  mem(precedes(TaskId, TaskName, SuccTList), Constraints), !,
  determine_est(TaskId, F_Time, SuccTsks, Tasks, UpTasks).
precedence_prop(_Task, _F_Time, Tasks, Tasks).

temporal_prop(_Task, _FTime, [], []).

temporal_prop(task(Task_F), FT, [task(Task_F1)|RTsks], [task(ModTask_F1)|UpTsks]) :-
  access_Fld_val([task_id, task_name]: Task_F, [TaskId,TaskName]),
  access_Fld_val([task_id, task_name]: Task_F1, [TaskId,TaskName1]),
  TaskName = TaskName1,
  !,
  assign_Fld_val([est]: Task_F1, [FT], ModTask_F1),
  temporal_prop(task(Task_F), FT, RTsks, UpTsks).

temporal_prop(task(Task_F), FT, RTsks, UpTsks).

Figure 4-9: Sample Prolog Predicates for Constraint Propagation

Other fields of the resources are now suitably modified, the modified fields are
updated resource load which keeps track of the total time the current resource is
utilized, the updated hours of the resource which indicate the time intervals in which
the resource is now available. The current resource with its modified attributes is
inserted into the resource pool, this is done by the predicate update_resource/7.
The difference between the three predicates defined for choose_update_resource/5
comes about in resource selection, which in the first two cases simply selects the first
resource from an ordered resource list. In the third case the predicate earliest_fin_rcs/4
tests the finishing time of each resource which can be allocated to the current task
and selects the one which finishes earliest. The other difference comes about in the
way updating of the resource list is done in the three cases, in the first and third
case of the predicate `update_resources` simply inserts the resource at the beginning of the resource list while in the second case a sorted resource list is maintained and the resource is inserted in the appropriate place.

![Diagram](image)

**Figure 4-10: Constraint Propagation in Jobshop Schedulers**

`choose_updt_resources/8` chooses and updates the resources to be allocated to a task `Tsk`. `select_tech/6` chooses the next technician based on `RcsHs` which is the resource selection heuristics. `determine_start_time/9` determines the exact start and finish times of the task which is a function of the earliest start time of the task `EST` and the availability slots of the operators `av(AvStart,AvEnd)`. Once the start and finish times of a task are determined, `update_tech_avail/5` modifies the remaining hours of availability of the operator. `update_resources/4` updates the resources after the current allocation.

After each iteration in the system, the time of availability of an operator is
updated. An example of time updation is shown in Figure 4-8 where the operator is previously available for the time ranges [(2,7),(8,10)], and is allocated to a task requiring 3 units of time starting at time=3, the updated availability hours of the operator will be [(2,3),(6,7),(8,10)] and he can be allocated now only during these periods.

Updation of resource availability is achieved by the predicate update_tech_avail/5.

4.7.3 Constraint Propagation
Partial ordering between different task are maintained in the system and earliest start times of tasks are updated by the constraint propagator after each iteration. The two kinds of constraint propagation done in the system are precedence propagation and temporal propagation. Once a task is scheduled, precedence propagation updates the earliest start time of all the tasks the current task precedes, provided those tasks do not have other preceding tasks. The updated earliest start time for these tasks is set to the finish time of the currently scheduled task.

Since tasks in the same job are mutually exclusive, the constraint propagator also adjusts the earliest start times of other unscheduled tasks of job, one of whose tasks is scheduled during the current iteration. This is done by the predicate temporal_prop/4. Figure 4-10 illustrates constraints which are propagated in a system once a task is scheduled.

Tasks $T_1^1, T_2^1, T_3^1$ can all start at the same time, task $T_1^1$ with a time duration of 4 is chosen from amongst the schedulable tasks. By temporal constraint propagation the earliest starting times of $T_2^1, T_3^1$ now become 6, the EST of task $T_4^1$ which was uninstantiated before, is now determined as 6.

precedence_propagation/5 tries to update the earliest start times of all the tasks which were constrained due to precedence constraints, after the task task(Task_F) has been scheduled and the new list of tasks is UpTsks1. temporal_prop/4 adjusts the EST's of other tasks having the same identifier as task(Task_F).

4.8 Empirical Results
The scheduling problem is NP Complete but simulation results indicate that for a carshop with about 8-10 workers and 30-35 jobs, the scheduler is able to give good
schedules as solutions.

Presenting a formal analysis of the scheduler performance is difficult because the solution method is empirical, and scheduling results were quite dependent on task requirements and operator availability. Also schedules produced by some resource allocation heuristics in the system like user preference allocation - in which the allocation of resources to tasks is done based on user preference, cannot be compared with the solutions produced by other heuristics where the objective is to minimize the cost or makespan of the schedule. User preference heuristic allocates resources to tasks as preferred by the user and does not guarantee a lower cost or minimal makespan. The presence of constraints simplifies the problem as the number of solutions become fewer, but the solutions produced by the different heuristics become less different. Hence an example with many constraints was not chosen to illustrate performance measures of different heuristics. Backtracking in the system however increases with an increase in the number of constraints.

A small problem with an output is presented in the current section. The best schedule with respect to both cost of the schedule and makespan was obtained by choosing the weighted shortest task heuristic and assigning the earliest finishing operators to each task. The results are followed by some statistics about results obtained by using different heuristics on the problem. The only precedence constraint in the system dictates that the fix_transmission task in car1 precedes the tune_up task.

Tasks:

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>Priority</th>
<th>EST</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix_transmission</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>tuneup</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>car2</td>
<td>fix_brakes</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>fix_transmission</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>replace_gasket</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>car3</td>
<td>replace_gasket</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>fix_brakes</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>car4</td>
<td>fix_transmission</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>change_muffler</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>car5</td>
<td>fix_fender</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>oil_change</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>fix_brakes</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Resources

<table>
<thead>
<tr>
<th>Operator</th>
<th>Aveil Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>al</td>
<td>(4,8),(9,12)</td>
</tr>
<tr>
<td>bert</td>
<td>(1,9),(11,16)</td>
</tr>
<tr>
<td>chip</td>
<td>(2,7),(9,16)</td>
</tr>
<tr>
<td>joe</td>
<td>(2,5),(7,12)</td>
</tr>
<tr>
<td>charles</td>
<td>(2,8),(10,14)</td>
</tr>
</tbody>
</table>
Operator Cost Time Data

<table>
<thead>
<tr>
<th>Op</th>
<th>Cst/Tim</th>
<th>Brakes</th>
<th>Gasket</th>
<th>Tuneup</th>
<th>Transm</th>
<th>Oil Change</th>
<th>Fix Fender</th>
</tr>
</thead>
<tbody>
<tr>
<td>al</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>bert</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>joe</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>charles</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

A Schedule

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>StartT</th>
<th>FinT</th>
<th>Operator</th>
<th>Flowtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix transmission</td>
<td>2</td>
<td>5</td>
<td>chip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tuneup</td>
<td>9</td>
<td>12</td>
<td>chip</td>
<td>12</td>
</tr>
<tr>
<td>car2</td>
<td>fix brakes</td>
<td>2</td>
<td>3</td>
<td>joe</td>
<td></td>
</tr>
<tr>
<td></td>
<td>replace gasket</td>
<td>3</td>
<td>5</td>
<td>joe</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fix transmission</td>
<td>7</td>
<td>10</td>
<td>joe</td>
<td>10</td>
</tr>
<tr>
<td>car3</td>
<td>fix brakes</td>
<td>5</td>
<td>6</td>
<td>al</td>
<td></td>
</tr>
<tr>
<td></td>
<td>replace gasket</td>
<td>6</td>
<td>9</td>
<td>bert</td>
<td>4</td>
</tr>
<tr>
<td>car4</td>
<td>change muffler</td>
<td>6</td>
<td>7</td>
<td>al</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fix transmission</td>
<td>9</td>
<td>11</td>
<td>al</td>
<td>6</td>
</tr>
<tr>
<td>car5</td>
<td>fix brakes</td>
<td>4</td>
<td>5</td>
<td>al</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fix fender</td>
<td>5</td>
<td>7</td>
<td>charles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oil change</td>
<td>7</td>
<td>8</td>
<td>charles</td>
<td>4</td>
</tr>
</tbody>
</table>

Statistics:

<table>
<thead>
<tr>
<th>TotalCost</th>
<th>Makespan</th>
<th>Avg Flowtime</th>
<th>Max Op Load</th>
<th>Avg Op Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>12</td>
<td>7.2</td>
<td>6</td>
<td>4.66</td>
</tr>
</tbody>
</table>

Cost and Makespan: Schedules optimized with respect to cost, also turned out to be optimized with respect to the makespan, in cases where hours of availability of different operators did not vary by more than 30%. For the schedule shown above the cost was 59, while the worst reported schedule had a cost of 73 units and a makespan of 16 time units. The schedules which allocated operators based on *earliest finish time operators* minimized the makespan.

Dispatch Contention Window Size: An optimal dispatch contention window is needed to minimize the overall cost of the schedule or the makespan. If the window size is 0, then dispatch contention occurs only between a few tasks - all
having the same earliest start times. With larger dispatch contention window size the backtracking in the system comes down and a better quality schedule resulted in a number of test cases. Dispatch window size of about 30-35% of the entire scheduling span produced better schedules with lower overall costs, makespan and reduced amount of backtracking, however in certain cases as the example problem a size less than 30% produced the best solution. The best solution for the problem previously described was found for DCWS=3, and had a cost of 59, with a makespan of 12. The average flowtime was 7.2, with maximum and average operator loads being 6 and 4.6 respectively.

Task Selection Heuristics: When a large number of schedules are possible weighted shortest time task first performed better than only priority or choosing the weighted longest task first. When the number of solutions are not very large, scheduling weighted longest time task first showed significantly less backtracking. The weighted shortest task first heuristic in combination with choosing the earliest finishing operator heuristic produced a lower flow time, a lower maximum task load and lower average task load on operators, for a large number of problems.

Operator Selection Heuristics: Choosing the earliest finishing operator first takes more execution time but yields solutions with lower costs and makespan. It has to evaluate the finish time of the tasks given all feasible operator combinations. Choosing operators with minimum load leads to more backtracking in the system compared to the previous heuristic.

Some sample data for the example problem:

<table>
<thead>
<tr>
<th>DCWS=0:</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskId</td>
<td>RstId</td>
<td>Cost</td>
<td>MakeSpan</td>
<td>Avg FT</td>
</tr>
<tr>
<td>Pri</td>
<td>min.ld</td>
<td>73</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>Pri</td>
<td>st.time</td>
<td>61</td>
<td>12</td>
<td>8.2</td>
</tr>
<tr>
<td>wstf</td>
<td>min.ld</td>
<td>63</td>
<td>16</td>
<td>10.4</td>
</tr>
<tr>
<td>wstf</td>
<td>st.time</td>
<td>63</td>
<td>12</td>
<td>7.4</td>
</tr>
<tr>
<td>wltf</td>
<td>min.ld</td>
<td>67</td>
<td>15</td>
<td>9.6</td>
</tr>
<tr>
<td>wltf</td>
<td>st.time</td>
<td>68</td>
<td>12</td>
<td>8.2</td>
</tr>
</tbody>
</table>
### DCWS=3:

<table>
<thead>
<tr>
<th>Task</th>
<th>Rcs Hs</th>
<th>Cost</th>
<th>MakeSpan</th>
<th>Avg FT</th>
<th>Max Load</th>
<th>Avg Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pri</td>
<td>min ld</td>
<td>73</td>
<td>16</td>
<td>10</td>
<td>8</td>
<td>4.8</td>
</tr>
<tr>
<td>Pri</td>
<td>ef time</td>
<td>61</td>
<td>12</td>
<td>8.2</td>
<td>6</td>
<td>4.6</td>
</tr>
<tr>
<td>wstf</td>
<td>min ld</td>
<td>66</td>
<td>16</td>
<td>10.2</td>
<td>10</td>
<td>6.2</td>
</tr>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>59</td>
<td>12</td>
<td>7.2</td>
<td>6</td>
<td>4.5</td>
</tr>
<tr>
<td>wltf</td>
<td>min ld</td>
<td>65</td>
<td>14</td>
<td>9</td>
<td>8</td>
<td>5.6</td>
</tr>
<tr>
<td>wltf</td>
<td>ef time</td>
<td>67</td>
<td>15</td>
<td>8.2</td>
<td>7</td>
<td>5.4</td>
</tr>
</tbody>
</table>

### DCWS=5:

<table>
<thead>
<tr>
<th>Task</th>
<th>Rcs Hs</th>
<th>Cost</th>
<th>MakeSpan</th>
<th>Avg FT</th>
<th>Max Load</th>
<th>Avg Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pri</td>
<td>min ld</td>
<td>66</td>
<td>14</td>
<td>9</td>
<td>8</td>
<td>5.8</td>
</tr>
<tr>
<td>Pri</td>
<td>ef time</td>
<td>61</td>
<td>12</td>
<td>8.2</td>
<td>6</td>
<td>4.6</td>
</tr>
<tr>
<td>wstf</td>
<td>min ld</td>
<td>68</td>
<td>16</td>
<td>9.8</td>
<td>9</td>
<td>6.0</td>
</tr>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>64</td>
<td>12</td>
<td>7</td>
<td>8</td>
<td>4.6</td>
</tr>
<tr>
<td>wltf</td>
<td>min ld</td>
<td>65</td>
<td>14</td>
<td>9</td>
<td>8</td>
<td>5.6</td>
</tr>
<tr>
<td>wltf</td>
<td>ef time</td>
<td>67</td>
<td>15</td>
<td>8.2</td>
<td>7</td>
<td>5.4</td>
</tr>
</tbody>
</table>

### DCWS=7:

<table>
<thead>
<tr>
<th>Task</th>
<th>Rcs Hs</th>
<th>Cost</th>
<th>MakeSpan</th>
<th>Avg FT</th>
<th>Max Load</th>
<th>Avg Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pri</td>
<td>min ld</td>
<td>66</td>
<td>14</td>
<td>9</td>
<td>8</td>
<td>5.8</td>
</tr>
<tr>
<td>Pri</td>
<td>ef time</td>
<td>61</td>
<td>12</td>
<td>8.2</td>
<td>6</td>
<td>4.6</td>
</tr>
<tr>
<td>wstf</td>
<td>min ld</td>
<td>62</td>
<td>16</td>
<td>9.6</td>
<td>10</td>
<td>6.0</td>
</tr>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>64</td>
<td>12</td>
<td>7</td>
<td>8</td>
<td>4.6</td>
</tr>
<tr>
<td>wltf</td>
<td>min ld</td>
<td>68</td>
<td>16</td>
<td>10</td>
<td>9</td>
<td>5.6</td>
</tr>
<tr>
<td>wltf</td>
<td>ef time</td>
<td>67</td>
<td>15</td>
<td>8.2</td>
<td>7</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Best Solutions if all operator are available from (1,16)

### DCWS=8:

<table>
<thead>
<tr>
<th>Task</th>
<th>Rcs Hs</th>
<th>DCWS</th>
<th>Cost</th>
<th>MakeSpan</th>
<th>Avg FT</th>
<th>Max Load</th>
<th>Avg Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>0</td>
<td>62</td>
<td>9</td>
<td>5.4</td>
<td>6</td>
<td>4.8</td>
</tr>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>3</td>
<td>60</td>
<td>9</td>
<td>5.0</td>
<td>6</td>
<td>4.6</td>
</tr>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>5</td>
<td>59</td>
<td>9</td>
<td>5.2</td>
<td>6</td>
<td>4.8</td>
</tr>
<tr>
<td>wstf</td>
<td>ef time</td>
<td>7</td>
<td>59</td>
<td>9</td>
<td>5.2</td>
<td>6</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Since there are more solutions possible now, longest task heuristic did not yield a better solution. A dispatch contention size window of about 30% of the entire schedule span produced the best solution. The backtracking significantly reduced
from the previous case for all heuristic combinations due to increased availability of operators.

Good schedules for problems depend on a host of factors, it is felt that a dispatch contention window size of about 25-30% of the schedule span, choosing weighted shortest task first and allocating earliest finishing operators to tasks yielded better solutions on a large range of problems.
4.9 Carshop Problem Traces

Carshop Schedule Problem:

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>Priority</th>
<th>EST</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix transmission</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>tuneup</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>car2</td>
<td>fix brakes</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>fix transmission</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>car3</td>
<td>fix gasket</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Resources

- Operator: Avail Time
  - al (4.7),(9.11)
  - bert (1.9),(11.16)
  - chip (2.7),(9.14)

Constraints

- precedes(car1, fix transmission, tuneup).

Operator Cost Time Data

<table>
<thead>
<tr>
<th>Op</th>
<th>Cost/Tim</th>
<th>Time Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>brakes  gasket tuneup transc AvTime</td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>1        2    3    3    2    (4.7),(9.11)</td>
</tr>
<tr>
<td>bert</td>
<td>1</td>
<td>3        3    3    2    3    (1.9),(11.16)</td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>3        3    3    3    3    (2.7),(9.16)</td>
</tr>
</tbody>
</table>

Sample Executions Provided for:

1. Schedule with
   - Dispatch Contention Window Size (DCWS) = 0 and
   - task ordering heuristic = wstf (weighted shortest task first)
   - operator ordering heuristic = min_load (op with minimum load assigned first)

2. Schedule with
   - Dispatch Contention Window Size (DCWS) = 0 and
   - task ordering heuristic = wstf (weighted shortest task first)
   - operator ordering heuristic = earliest_fin_op (op with minimum completion time assigned first)

3. Schedule with
   - Dispatch Contention Window Size (DCWS) = 5 and
   - task ordering heuristic = wstf (weighted shortest task first)
• operator ordering heuristic = min_load (op with minimum load assigned first)

4. Schedule involving backtracking on task orderings
Schedule using DCWS=0, wstf, min_load heuristics

Iteration 1:

Time=0

- Earliest schedulable tasks expressed as (Car,Task,EST) are ((Car1,transm,0), (Car2,brakes,0), (Car2,transm,0))^0

- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>Pr*Wt=</th>
<th>(Car2,brakes)</th>
<th>(Car1,transm)</th>
<th>(Car2,transm)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td></td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bert</td>
<td></td>
<td></td>
<td>(5,7),(9,11)</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>chip</td>
<td></td>
<td></td>
<td>(1,9),(11,16)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

New EST's

<table>
<thead>
<tr>
<th></th>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>0</td>
<td>5^1</td>
<td>5</td>
</tr>
</tbody>
</table>

Iteration 2:

Time=0

- Earliest schedulable tasks = ((Car1,transm,0))^3

- No heuristic required to break ties between tasks.

- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>Pr*Wt=</th>
<th>(Car1,transm)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>ST</td>
<td>FT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bert</td>
<td>1</td>
<td>5</td>
<td>(5,9),(11,16)</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>chip</td>
<td></td>
<td></td>
<td>(2,7),(9,10)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>al</td>
<td></td>
<td></td>
<td>(6,7),(9,11)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

New EST's

<table>
<thead>
<tr>
<th></th>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Unscheduled Tasks

<table>
<thead>
<tr>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>Task</td>
<td>EST</td>
</tr>
<tr>
<td>0</td>
<td>transm</td>
<td>5</td>
</tr>
<tr>
<td>x^2</td>
<td>tuneup</td>
<td></td>
</tr>
</tbody>
</table>

^0 Precedence Constraint: (car1,tuneup) is not in this set due to precedence constraint

^1 Temporal Propagation: car2 is scheduled from (4,5); so is available only after T=5.

^2 Temporal Partial Ordering: transm precedes tuneup in car1, so EST of (car1,transm) is not known.

^3 Dispatch Contention Window Size: Since DCWS=0, only one task at time T=0 competes for dispatch.
Iteration 3:

Time=5

- Earliest schedulable tasks = \{(Car2,transm,5),(Car3,gasket,5)\}^0\)

- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>Pr<em>Wt-- (Car3,gasket) 3</em>3=9</th>
<th>(Car2,transm) 1*2=2</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops ST FT ST FT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chip 9 12 (5,7),(12,16) 0 3 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>al   5 7 (5,7),(9,11) 1 1 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bert 6 9 (6,9),(11,16) 5 5 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

New EST's
<table>
<thead>
<tr>
<th>EST</th>
<th>Carl1</th>
<th>Carl2</th>
<th>Carl3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>6</td>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

Unscheduled Tasks
<table>
<thead>
<tr>
<th>Carl1</th>
<th>Task</th>
<th>Carl2</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>tuneup</td>
<td>Task</td>
<td>transm</td>
</tr>
</tbody>
</table>

Iteration 4:

Time=5^1

- Earliest schedulable tasks = \{(Car2,transm,5)\}

- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>(Car2,transm)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST FT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>al 5 7 (5,7),(9,11) 1 3 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bert</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

New EST's
<table>
<thead>
<tr>
<th>EST</th>
<th>Carl1</th>
<th>Carl2</th>
<th>Carl3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

Unscheduled Tasks
<table>
<thead>
<tr>
<th>Carl1</th>
<th>EST Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>tuneup</td>
</tr>
</tbody>
</table>

^0 Both (car2,transm) and (car3,gasket) are available at time T=5 and compete for dispatch
^1 Time: This is the minimum of all the EST's found in the previous iteration
Iteration 5:

Time=6

- Earliest schedulable tasks = { (Car1.tuning,6) }

- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>1 Ops(LOAD)</th>
<th>Ordered Tasks(Weights)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Car1.tuning)</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>chip</td>
<td></td>
</tr>
<tr>
<td>al</td>
<td></td>
</tr>
</tbody>
</table>

- No tasks left to be Scheduled
- No constraints violated

Schedule

<table>
<thead>
<tr>
<th>Car</th>
<th>Car, Task, StartT, FinT, Operator, Flowtime, Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix, transmission, 1, 6, bert, 5</td>
</tr>
<tr>
<td>car2</td>
<td>tuneup, 11, 16, bert, 16, 6</td>
</tr>
<tr>
<td>car3</td>
<td>fix, brakes, 4, 5, al, 4</td>
</tr>
<tr>
<td></td>
<td>fix, transmission, 5, 7, al, 7, 8</td>
</tr>
<tr>
<td></td>
<td>fix, gasket, 9, 12, chip, 7, 6</td>
</tr>
</tbody>
</table>

Statistics:

<table>
<thead>
<tr>
<th>TCost</th>
<th>Makespan</th>
<th>Avg Flowtime</th>
<th>Max Op Load</th>
<th>Avg Op Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>16</td>
<td>10</td>
<td>10</td>
<td>5.33</td>
</tr>
</tbody>
</table>

\(^0\) Flowtime: This is the difference between Completion time and Arrival time of a job ie. (12-5) = 7

\(^1\) Makespan: This is the difference between Completion time of the last job from the starting scheduling time.
Schedule using DCWS=0, wstf, earliest_fin_op heuristics

Iteration 1:
Time=0

- Earliest schedulable tasks expressed as (Car,Task,EST) are 
  
  $\{(\text{Car1,transm,0}), (\text{Car2,brakes,0}), (\text{Car2,transm,0})\}$

- Operator selection $^0$:

<table>
<thead>
<tr>
<th>Ops</th>
<th>(Car2,brakes)</th>
<th>(Car1,transm)</th>
<th>(Car2,transm)</th>
<th>Avail</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pr<em>Wt=2</em>5≤10</td>
<td>2*5≤10</td>
<td>4*2≥8</td>
<td>1*2=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>5</td>
<td>(4,7)/(9,11)</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>bert</td>
<td>1</td>
<td>4</td>
<td>(1,2)/(11,16)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>5</td>
<td>(2,7)/(9,16)</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

- Task Ordering(on weights), Earliest Finishing Op, Allocation:

<table>
<thead>
<tr>
<th>Ops</th>
<th>Ordered Tasks(Weights)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pr<em>Wt=2</em>5≤10</td>
<td>(Car2,brakes)</td>
<td>(Car1,transm)</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>Ops</td>
<td>ST</td>
<td>FT</td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car1</td>
</tr>
<tr>
<td>EST</td>
</tr>
</tbody>
</table>

Iteration 2:
Time=0

- Earliest schedulable tasks = $\{(\text{Car1,transm,0})\}$

- No heuristic required to break ties between tasks.

- Operator selection $^3$:

<table>
<thead>
<tr>
<th>Ops</th>
<th>(car1,transm)</th>
<th>Avail</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>bert</td>
<td>4</td>
<td>9</td>
<td>(4,9)/(11,16)</td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>6</td>
<td>(4,7)/(9,11)</td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>5</td>
<td>(2,7)/(9,16)</td>
</tr>
</tbody>
</table>

$^0$Operator Selection: The selected operator is able to complete the task within all constraints.

$^1$Task Selection: Task with the highest rating is chosen to be scheduled first.

$^2$(car2,brakes): bert finishes earliest at time $T=4$.

$^3$Breaking Ties: If many operators finish at the same time (also fastest), the one with minimum load is chosen.

$^4$(car1,transm): chip finishes earliest at time $T=5$. 
1 Ops

<table>
<thead>
<tr>
<th>(Car1,transm)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>FT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>5</td>
<td>(5,7),(9,16)</td>
<td>0</td>
</tr>
<tr>
<td>bert</td>
<td>(4,9),(11,16)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>al</td>
<td>(4,7),(9,11)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TCost</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Temporal constraint propagation on unscheduled tasks:

New EST's

<table>
<thead>
<tr>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Iteration 3:

Time=4

- Earliest schedulable tasks = \{ (Car2,transm,4) \}

- Operator selection:

<table>
<thead>
<tr>
<th>(Car2,transm)</th>
<th>Ops</th>
<th>ST</th>
<th>FT</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>chip</td>
<td>9</td>
<td>12</td>
<td>(5,7),(9,16)</td>
<td>6</td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>9</td>
<td>(4,7),(9,11)</td>
<td>8</td>
</tr>
<tr>
<td>bert</td>
<td>(4,9),(11,16)</td>
<td>8</td>
<td>(4,9),(11,16)</td>
<td>4</td>
</tr>
</tbody>
</table>

Task Ordering(on weights), Earliest Finishing Op, Allocation:

1 Ops(Load)

<table>
<thead>
<tr>
<th>(Car2,transm)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>ST</td>
<td>FT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>al</td>
<td>1</td>
<td>6</td>
<td>(6,7),(9,11)</td>
<td>0</td>
</tr>
<tr>
<td>bert</td>
<td>(4,9),(11,16)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>chip</td>
<td>(5,7),(9,16)</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>TCost</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Temporal constraint propagation on unscheduled tasks:

New EST's

<table>
<thead>
<tr>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Unscheduled Tasks

<table>
<thead>
<tr>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>Task</td>
<td>EST</td>
</tr>
<tr>
<td>5</td>
<td>tuneup</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^0\text{Time: This is the minimum of all the EST's found in the previous iteration}\)
Iteration 4:

**Time=5**

- Earliest schedulable tasks = \{(Car3,tuneup,5),(Car3,gasket,5)\}

<table>
<thead>
<tr>
<th>Pr*Wt---</th>
<th>(Car3,gasket) 3*3=9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>ST</td>
</tr>
<tr>
<td>bert</td>
<td>5</td>
</tr>
<tr>
<td>al</td>
<td>9</td>
</tr>
<tr>
<td>chip</td>
<td>5,7</td>
</tr>
</tbody>
</table>

- Operator selection:

- Task Ordering(on weights), Earliest Finishing Op, Allocation:

<table>
<thead>
<tr>
<th>Ops(Load)</th>
<th>--- Ordered Tasks(Weights)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pr*Wt---</td>
<td>(Car3,gasket) 3*3=9</td>
</tr>
<tr>
<td>Ops</td>
<td>ST</td>
</tr>
<tr>
<td>bert</td>
<td>5</td>
</tr>
<tr>
<td>chip</td>
<td>9</td>
</tr>
<tr>
<td>al</td>
<td>9</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

| Unscheduled Tasks |
|---|---|
| Carl | EST Task |
| 5 | tuneup |

Iteration 5:

**Time=5**

- Earliest schedulable tasks = \{(Car1,tuneup,5)\}

<table>
<thead>
<tr>
<th>(Car1,tuneup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
</tr>
<tr>
<td>bert</td>
</tr>
<tr>
<td>al</td>
</tr>
<tr>
<td>chip</td>
</tr>
</tbody>
</table>

- Operator selection:

- Task Ordering(on weights), Earliest Finishing Op, Allocation:

<table>
<thead>
<tr>
<th>Ops(Load)</th>
<th>--- Ordered Tasks(Weights)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Car1,tuneup)</td>
<td>Avail</td>
</tr>
<tr>
<td>Ops</td>
<td>ST</td>
</tr>
<tr>
<td>chip</td>
<td>9</td>
</tr>
<tr>
<td>al</td>
<td>9</td>
</tr>
<tr>
<td>bert</td>
<td>9</td>
</tr>
</tbody>
</table>

- No tasks left to be Scheduled
- No constraints violated

---

*Availability Constraint: al cannot be assigned to (car1,tuneup) in his available hours.*
### Schedule

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>StartT</th>
<th>FinT</th>
<th>Operator</th>
<th>Flowtime</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix_transmission</td>
<td>2</td>
<td>5</td>
<td>chip</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>tuneup</td>
<td>9</td>
<td>12</td>
<td>chip</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>car2</td>
<td>fix_brakes</td>
<td>1</td>
<td>4</td>
<td>bert</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>fix_transmission</td>
<td>4</td>
<td>6</td>
<td>al</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>car3</td>
<td>fix_gasket</td>
<td>5</td>
<td>8</td>
<td>bert</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>

### Statistics:

<table>
<thead>
<tr>
<th>TCost</th>
<th>Makespan</th>
<th>Avg Flowtime</th>
<th>Max Op Load</th>
<th>Avg Op Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>4.67</td>
</tr>
</tbody>
</table>
Schedule using DCWS=5, wstf, min_load heuristics

**Iteration 1:**

- Earliest schedulable tasks are \{\langle Car1,transm,0 \rangle, \langle Car2,brakes,0 \rangle, \langle Car2,transm,0 \rangle, \langle Car3,gasket,5 \rangle\} \textsuperscript{0}
- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>Pr*Wt→</th>
<th>(Car2,brakes) (2^*5=10)</th>
<th>(Car3,gasket) (3^*3=9)</th>
<th>(Car1,transm) (4^*2=8)</th>
<th>(Car2,transm) (1^*2=2)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>al</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5,7</td>
<td>9,11</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>chip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,9</td>
<td>11,16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2,7</td>
<td>9,16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car1</td>
</tr>
<tr>
<td>EST</td>
</tr>
</tbody>
</table>

**Iteration 2:**

- Earliest schedulable tasks =\{\langle Car1,transm,0 \rangle, \langle Car2,transm,5 \rangle, \langle Car3,gasket,5 \rangle \} \textsuperscript{1}
- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>Pr*Wt→</th>
<th>(Car3,gasket) (3^*3=9)</th>
<th>(Car1,transm) (4^*2=8)</th>
<th>(Car2,transm) (1^*2=2)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ops</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td></td>
</tr>
<tr>
<td>bert</td>
<td>5^2</td>
<td>8</td>
<td></td>
<td></td>
<td>1,5</td>
<td>6,9</td>
<td>(11,16)</td>
</tr>
<tr>
<td>chip</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td>2,7</td>
<td>9,16</td>
<td>0</td>
</tr>
<tr>
<td>sl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5,7</td>
<td>9,11</td>
<td></td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car1</td>
</tr>
<tr>
<td>EST</td>
</tr>
</tbody>
</table>

---

\textsuperscript{0} Prcedence Constraint: (car1,tuneup) is still not in this set due to precedence constraint.

\textsuperscript{1} Earliest schedulable tasks: Since the time window is 5 (car3,gasket) with EST 5 is also schedulable now.

\textsuperscript{2} Time Window: is \((0,4+5)\) where 0 is the minimum EST of the Unscheduled tasks, 5 is the DCWS.

\textsuperscript{3} Start Time: is determined by taking the maximum between Task EST, and lowerbound of operator availability time.
Iteration 3:

Time=\(0,5\)

- Earliest schedulable tasks = \{(\text{Car2.transm},5), (\text{Car1.transm},0)\}
- Task Ordering (on weights), Operator Ordering (on load), Allocation:
  1 Ops (Load) — Ordered Tasks (Weights)

<table>
<thead>
<tr>
<th>Pr=Wt—</th>
<th>(Car1,transm) (4^*2=8)</th>
<th>(Car2,transm) (1^*2=2)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cps</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td>((5,7),(9,16))</td>
<td>0</td>
</tr>
<tr>
<td>chip</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>((5,7),(9,11))</td>
<td>1</td>
</tr>
<tr>
<td>al</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>((1,5),(8,9),(11,16))</td>
<td>3</td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Iteration 4:

Time=\(5,10\)

- Earliest schedulable tasks = \{(\text{Car2.transm},5), (\text{Car1.tuneup},5)\}
- Task Ordering (on weights), Operator Ordering (on load), Allocation:
  1 Ops (Load) — Ordered Tasks (Weights)

<table>
<thead>
<tr>
<th>Pr=Wt—</th>
<th>(Car2,transm) (3^*1=3)</th>
<th>(Car1,tuneup) (1^*2=2)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cps</td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td>((9,11))</td>
<td>1</td>
</tr>
<tr>
<td>chip</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td>((5,7),(9,16))</td>
<td>3</td>
</tr>
<tr>
<td>al</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>((1,5),(8,9),(11,16))</td>
<td>3</td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
<th>Car1</th>
<th>Car2</th>
<th>Car3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

| Unscheduled Tasks |
|-------------------|--------|
| Car1              | EST: 5 |
| Task: tuneup      |       |
| Car2              | EST: 5 |
| Task: transm      |       |

\(^0\) Start Time: \(ST=2\) because \(\max(0,2)=2\), here 0 is the task EST and lowerbound of closest availability range ie. \((2,7)\) is 2

\(^1\) Time Window: This changes now to \((5,10)\) as unscheduled tasks have a minimum EST of 5.

\(^2\) Earliest schedulable tasks: EST of (Car1,tuneup) is determined now; and is within the current contention time window.
Iteration 5:

Time = 5

- Earliest schedulable tasks = \{(Car1, tuneup, 5)\}

- Task Ordering(on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>(Car2, transm)</th>
<th>ST</th>
<th>FT</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>bert</td>
<td>11</td>
<td>16</td>
<td>(1,5),(8,9)</td>
<td>3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>chip</td>
<td></td>
<td></td>
<td>(5,7),(9,16)</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>al</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

TCost: 26

- No tasks left to be Scheduled
- No constraints violated

Schedule

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>StartT</th>
<th>FinT</th>
<th>Operator</th>
<th>Flowtime</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>car1</td>
<td>fix_transmission</td>
<td>2</td>
<td>5</td>
<td>chip</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tuneup</td>
<td>11</td>
<td>16</td>
<td>bert</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>car2</td>
<td>fix_brakes</td>
<td>4</td>
<td>5</td>
<td>al</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fix_transmission</td>
<td>5</td>
<td>7</td>
<td>al</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>car3</td>
<td>fix_gasket</td>
<td>5</td>
<td>8</td>
<td>bert</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Statistics:

<table>
<thead>
<tr>
<th>TCost</th>
<th>Makespan (^1)</th>
<th>Avg Flowtime</th>
<th>Max Op Load</th>
<th>Avg Op Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>1.6</td>
<td>8.57</td>
<td>8</td>
<td>4.67</td>
</tr>
</tbody>
</table>

\(^0\)Start Time: Increasing DCWS tends to allow higher rated tasks to be scheduled earlier.

\(^1\)Makespan: This is the lowest of all three examples.
Example of Backtrack on Task Dispatch:

<table>
<thead>
<tr>
<th>Car</th>
<th>Tasks</th>
<th>Priority</th>
<th>EST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car1</td>
<td>brakes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>gasket</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Car2</td>
<td>tuneup</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Avail Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>chip</td>
<td>(2.5),(8,10)</td>
</tr>
<tr>
<td>bert</td>
<td>(1.4),(5,8)</td>
</tr>
<tr>
<td>ai</td>
<td>(5,10)</td>
</tr>
</tbody>
</table>

Operator Cost Time Data: 0

<table>
<thead>
<tr>
<th>Op</th>
<th>Cat/Tim</th>
<th>Time Duration</th>
<th>AvTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>chip</td>
<td>2</td>
<td>brakes 2, gasket 3</td>
<td>(2,5),(8,10)</td>
</tr>
<tr>
<td>bert</td>
<td>1</td>
<td>tuneup 4</td>
<td>(1,4),(5,8)</td>
</tr>
<tr>
<td>ai</td>
<td>4</td>
<td>2</td>
<td>(5,10)</td>
</tr>
<tr>
<td>wstf Wt</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Backtracking on Task Orderings, Schedule using DCWS=0, wstf, min_load

Iteration 1:

Time=1

- Earliest schedulable tasks expressed are \{(Car1, brakes, 1), (Car1, gasket, 1)\}

- Task Ordering(on weights), Operator Ordering(on load)\(^1\) Allocation:

<table>
<thead>
<tr>
<th>Op</th>
<th>(Car1, brakes)</th>
<th>(Car1, gasket)</th>
<th>Avail</th>
<th>Load0</th>
<th>Load0</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ST</td>
<td>FT</td>
<td>ST</td>
<td>FT</td>
<td>(4.5),(8,10)</td>
<td>0</td>
</tr>
<tr>
<td>chip</td>
<td>2</td>
<td>4</td>
<td>(1,4),(5,8)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car1</td>
</tr>
<tr>
<td>EST</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unscheduled Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car1</td>
</tr>
<tr>
<td>EST</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

\(^0\)Cost Information: Cost of assignment for forbidden operator task assignment is considered infinite and not given.

\(^1\)Forbidden Op: Since ai is not eligible to work on the presently schedulable tasks, he is omitted.
Iteration 2:

Time=4

- Earliest schedulable tasks = {{Car1.gasket,4}}
- No heuristic required to break ties between tasks.

- Task Ordering (on weights), Operator Ordering (on load), Allocation:

<table>
<thead>
<tr>
<th>Ops(Load)</th>
<th>ST</th>
<th>FT</th>
<th>(Car1.gasket)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>bert 1</td>
<td>-</td>
<td>-</td>
<td>(1,4),(5,8)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>chip 2</td>
<td>-</td>
<td>-</td>
<td>(8,10)</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

- New Rating calculation using random weights factors

Wt factors used: 2.3 for gasket, 1 for brakes as it was schedulable previously.

New Wts are 2.3*3=6.9 for gasket, 5 (unchanged) for brakes.

Iteration 1: 3

Time=1

- Earliest schedulable tasks expressed are {{Car1.brakes,1}, (Car1.gasket,1)}

- Task Ordering (on weights), Operator Ordering (on load) Allocation:

<table>
<thead>
<tr>
<th>Ops(Load)</th>
<th>ST</th>
<th>FT</th>
<th>(Car1.gasket)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>chip</td>
<td>2</td>
<td>5</td>
<td></td>
<td>(8,10)</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>bert</td>
<td></td>
<td></td>
<td>(1,4),(5,8)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

- Temporal constraint propagation on unscheduled tasks:

<table>
<thead>
<tr>
<th>New EST's</th>
<th>Car1</th>
<th>Car2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Unscheduled Tasks

<table>
<thead>
<tr>
<th>Car1</th>
<th>Car2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST</td>
<td>Task</td>
</tr>
<tr>
<td>5</td>
<td>brakes</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

---

Unschedulable Task: (Car1.gasket) is unschedulable now as no operator can finish it in his available hours.

1 bert's availability: bert takes 4 time units to fix (Car1.gasket) but is available for only 3 unit periods (1,4),(5,8).

2 chip's availability: chip takes 3 time units to fix the gasket but is available only from (8,10).

3 Iteration: The schedule is started all over again with no scheduled tasks.

4 BackTrack Ratings: Weights of the previously unscheduled tasks are changed by a random factor.
Iteration 2:

Time=5

- Earliest schedulable tasks = \{ (\text{car1.brakes,5} ) \}
- No heuristic required to break ties between tasks.

- Task Ordering 0 (on weights), Operator Ordering(on load), Allocation:

<table>
<thead>
<tr>
<th>Ops(Load)</th>
<th>(Car1.brakes)</th>
<th>Avail</th>
<th>Load1</th>
<th>Load2</th>
<th>OpCost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>FT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bert 1</td>
<td>5</td>
<td>8</td>
<td>(1.4)</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>chip 2</td>
<td></td>
<td></td>
<td></td>
<td>(8.10)</td>
<td>3</td>
</tr>
<tr>
<td>TCost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Schedule continues for (car2,tuneup) . . . .

4.10 Summary

In this chapter a scheduling cluster called the jobshop scheduler has been described and one instance of it called the carshop scheduler has been presented in detail. Each schedule consists of a series of resource allocations to tasks, and the objective of the schedule is to try and make "good" assignments of resources to all tasks. Specifically for the carshop scheduler different heuristics for task dispatch and operator allocation were discussed. The scheduler is a generate and test scheduler which tries to find a good schedule with respect to cost of the schedule or makespan of the schedule depending on the secondary objective criterion chosen by the user. Issues in backtracking over schedules and partial schedules were discussed and intelligent backtracking was achieved by preferentially increasing the weights of unscheduled tasks and tasks which precede them due to precedence constraint. Several schedule iterations are tried and the best solution is finally reported to the user, the strategies help in generating only potentially good and feasible solutions. Good schedules for problems depend on a host of factors, it is felt that a dispatch contention window size of about 25-30% of the schedule span, choosing weighted shortest task first and allocating earliest finishing operators to tasks yielded better solutions on a large range of problems.

---

0Task Ordering: The new weight factors cause (car1.brakes) to be scheduled later than (car1.gasket).
1Schedulable Task: bert can now be assigned (Car1.brakes).
2chip's assignment: The prior assignment of chip to (car1.gasket) helped complete the schedule.
Chapter 5

Transportation Schedulers

"All I ask is a tall ship...and a star to steer her by..." You could feel the wind at your back, about you... the sounds of the sea beneath you. And even if you take away the wind and the water, it's still the same. The ship is yours...you can feel her...and the stars are still there.


5.1 Introduction

During the last twenty years, many papers have been devoted to the development of optimization and approximation algorithms for vehicle routing and scheduling problems [GCJS57], [DR59], [FR76], [App71], [MH71], [Ron83]. The interest is due to the practical need for effective and efficient methods for physical distribution as well as to the intriguing nature of the underlying combinatorial optimization models. The essential notion of transportation scheduling is to efficiently allocate transportation assets or resources (e.g. planes, ships, trucks) to some movements (e.g. bulk cargo, passengers) over time intervals in a manner so as to minimize the total transportation cost which is an indicator of the "goodness" of the schedule. All allocations are subject to constraints which dictate if specific assignments of transporters to movements are feasible or not. There are two primary kinds of
transportation problem which have attracted attention, the first one being land based vehicle routing problems, and the other one being ocean tanker scheduling. The variety in each of these scheduling problems comes about due to the various combinations of constraints one must satisfy in each of the scheduling problems. We develop the transportation cluster based on generic design principles proposed in Chapter 3. One of the contributions of this research is to present a unified approach for handling both these kinds of transportation problem.

This chapter is organized as follows - the land based vehicle routing problem and ocean tanker problem are presented in separate sections 5.2 and 5.3 respectively. In these sections the problem is defined and an integer linear program (ILP) formulation for each of the land based vehicle routing problem and ocean transport scheduling problem is presented. From the formulations one can see how even the simplest of problems gives rise to a large number of variables and it is computationally very hard if not infeasible to solve the problems. These problems also have relaxed temporal constraints, whereas many real-life problems being solved currently require strict temporal constraints.

Our solution to the transportation problem is to specialize the generic generate and test scheduling algorithm proposed in Chapter 3 to the transportation cluster. Following the philosophy of the research; tasks, resources and constraints are identified for the current problem. Each one of the tasks, and resources are represented as objects and described in greater detail in Subsections 5.4.1, 5.4.2, 5.4.4, routing which is an integral part of dispatch schedulers is also discussed in detail in Subsection 5.7.4. Heuristics are used to guide the schedule and these are discussed in the sections showing task and resource schedule heuristics definitions. Finally some examples and results are presented in Section 5.8.

The generate and test scheduling example used for the transportation cluster proceeds to systematically select a destination for delivery, chooses an appropriate transport vessel, the selected vessel is allocated to the destination and its capacity, availability are suitably updated. Constraints resulting from the allocation are then propagated. The same cycle continues until all demands to various destinations are satisfied or a situation where no resource can be allocated to the current destination
is reached. The exact algorithm is presented in Section 5.7. Infeasibilities are handled by backtracking over the schedule generation phase, and the best solution in a given number of iterations is selected as the final schedule.

In order to develop a successful transportation scheduler information needs to be represented and manipulated at three levels. At the first level, is the real-life problem situation which may contain many aspects that are not relevant for the selection of a solution method. At the second level in each of the transportation problems there is an abstract problem type. It is obtained from the real-life problem by determining and modeling the relevant entities in terms of decisions, objectives and constraints. At the third level, there are the algorithms. One that appears to be suitable at hand is selected. Desrochers, Lenstra and Savelbergh [DLS90] provide a classification scheme for vehicle routing and scheduling problems. J.K. Lenstra and Rinnoy Kan [LK81] have also analyzed the complexity of some vehicle routing problems.

A brief description and mathematical formulation of land based vehicle routing and ocean transport schedulers are now presented.

5.2 Land Based Vehicle Routing Problem

The land based vehicle routing problem [BG81], [LK81], [Mag81] can be defined as follows: A set of customers each with a known location and known requirement for some commodity, are to be supplied from a set of depots by a set of delivery vehicles of known capacity. The objective of the solution may be stated as cost minimization (distribution costs and vehicle or depot acquisition cost), or service improvement (increasing distribution capacities, reducing distribution time and related network design issues). It is required to design the routes for the vehicles subject to the following constraints:

1. The requirements of all the customers must be met. This constraint ensures a balance between demand and supply, and the supply should be at least equal to the demand in the system.
2. The node constraints of vehicles must not be violated. This constraint ensures that some of the vehicles are not overused and that the number of nodes/customers allocated to each vehicle does not exceed some predetermined number.

3. The cost (alternatively time or distance) constraints of vehicles must not be violated. This constraint dictates that the total cost of each vehicle to complete its journey must not exceed some pre-determined level. Also the total distance traveled by a vehicle during a schedule time, or the time of travel are constrained below a fixed limit.

4. The load constraints of the vehicle must not be violated. This constraint ensures that the total load allocated to each vehicle does not exceed its capacity.

Depending on the number of depots the problem must be classified as a single depot, or a multiple depot problem. Several subproblems and their variations may be formulated by considering various possible combinations of the four constraints discussed earlier. In order to simplify the problem, if it is assumed that the fleet consists of M vehicles of sufficiently large capacity, so that constraint (1), (2), (3) and (4) can be ignored and the problem reduces to the M-traveling Salesman problem (M-TSP) [Ber85]. In its simplest version, if it is further assumed that there is only one vehicle of very large capacity then the problem reduces to the well known traveling salesman problem (TSP). It is for this reason that the formulation for TSP is taken as a core model for the development of the mathematical formulations for the more complicated cases. Therefore most of the mathematical formulations of the vehicle routing problems are variants and/or extensions of the well known traveling salesman problem (TSP). A vehicle routing problem can be formulated as a dynamic program or as an Integer Linear Programming problem, however the integer linear program is more used because of its simplicity.

A truck dispatch problem is an instance of a land based vehicle routing problem. This commonly occurs in the petroleum industry where gas stations have to be delivered supplies. The objective in a truck dispatch problem is to allocate trucks
on various land routes which will ship material to different destinations. The cost function in the simplest case is proportional to the distance traveled during delivery.

1. A set of *n* station points \( P_i \ (i = 1, 2, \cdots, n) \) to which deliveries are made from point \( P_o \) called the terminal point.

2. A *Distance Matrix* \([D]\) where \( d_{ij} \) is given which specifies the distance between every point \((i = 1, 2, \cdots, n)\) such that \( d_{ij} = d_{ji} \).

3. A *delivery vector* \((Q) = (q_i)\) is given which specifies the amount \( q_i \) to be delivered to every point \( P_i \ (i = 1, 2, \cdots, n) \).

4. The truck capacity is \( C \) where \( C \geq \max q_i \).

5. If \( x_{ij} = x_{ji} = 1 \) is interpreted to mean that points \( P_i \) and \( P_j \) are paired \((i, j = 0, 1, \cdots, n)\), deliveries to these are made by the same truck. \( x_{ij} = x_{ji} = 0 \) means that the points are not paired. One obtains the condition:

\[
\sum_{j=0}^{n} x_{ij} = 1 \quad \text{for } i = 1, 2, \cdots, n
\]

since every point \( P_i \) is connected with \( P_o \) or at most one other point \( P_j \).

6. The problem is to find those values of \( x_{ij} \) which make the total distance

\[
D = \sum_{i,j=0}^{n} d_{ij} x_{ij}
\]

a minimum under the constraints given before.

One of the earliest references for the truck dispatch problem is a paper by Dantzig and Ramser [DR59], who first developed a heuristic approach using linear programming ideas and aggregation of nodes. In the above formulation each customer is served by one and only one vehicle.

The place of origin of any of the vehicles is \( P_o \), in the statement of the truck dispatch problem, hence it is called a single depot vehicle routing problem. Any vehicle routing problem with multiple origins is called a multiple depot vehicle routing problem.
5.2.1 Single Depot Vehicle Routing Problem

The formulation of a single depot vehicle routing problem is presented here. We assume that the vehicles have only the capacity and maximum cost (time or distance constraints).

The variables in the system are defined first

\[ V = \text{Number of Vehicles} \]
\[ P_k = \text{Capacity of vehicle } k \]
\[ T_k = \text{Maximum cost allowed for route of vehicle } k \]
\[ c_{ij} = \text{Cost of travelling from } i \text{ to } j \]
\[ Q_i = \text{Demand at node } i, \ Q_n = 0 \]
\[ x_{ijk} = 1 \text{ if pair } i, j \text{ is in the route of vehicle } k, \ 0 \text{ otherwise} \]

The problem is to

\[ \min Z = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{V} c_{ij} x_{ijk} \]

\[ \sum_{i=1}^{N} \sum_{k=1}^{V} x_{ijk} = 1 \text{ for } j = 1, 2, \ldots, N - 1 \quad (5.1) \]

\[ \sum_{j=1}^{N} \sum_{k=1}^{V} x_{ijk} = 1 \text{ for } i = 1, 2, \ldots, N - 1 \quad (5.2) \]

The route continuity is expressed by constraint 5.3, this states that the number of vehicles allocated to a route going into node \( h \) is exactly equal to the number of vehicles leaving node \( h \). The capacity constraint of a vehicle \( P_k \) in 5.4 dictates the maximum amount of material it can supply to all the nodes occurring in a route through which \( P_k \) travels.

\[ \sum_{i=1}^{N} x_{ikh} - \sum_{j=1}^{N} x_{hjk} = 0 \text{ for } k = 1, 2, \ldots, V, \]
\[ h = 1, 2, \ldots, N \quad (5.3) \]

\[ \sum_{i=1}^{N} Q_i \sum_{j=1}^{N} x_{ijk} \leq P_k \text{ for } k = 1, 2, \ldots, V \quad (5.4) \]
The total route constraint which limits the cost incurred by any given vehicle is given by 5.5

\[ \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} x_{ijk} \leq T_k \quad \text{for } k = 1, 2, \ldots, V \quad (5.5) \]

The inequations 5.6, 5.7 make sure that the vehicle availability is not exceeded. The equations 5.9, 5.11 make sure that subtour assignments do not satisfy the previous constraints. Several other alternatives for defining $S$ have been proposed in the literature and here $S$ leads to $2^N$ subtour breaking constraints which is really large for a large $N$. Miller, Tucker and Zemlin [MTZ85] have proposed a alternative set $S$ representation which leads to only $(N^2 - 3N + 2)$ constraints.

\[ \sum_{j=1}^{N-1} x_{Njk} \leq 1 \quad \text{for } k = 1, 2, \ldots, V \quad (5.6) \]

\[ \sum_{i=1}^{N-1} x_{iNk} \leq 1 \quad \text{for } k = 1, 2, \ldots, V \quad (5.7) \]

\[ x_{ijk} = 0 \text{ or } 1 \quad \text{for all } i, j, k \quad (5.8) \]

\[ x_{ijk} \in S \quad (5.9) \]

\[ S = \left\{ (x_{ijk}) : \sum_{i \in B} \sum_{j \in B} x_{ijk} \geq |B| - 1 \quad (5.10) \right. \]

\[ \text{for every nonempty subset } B \text{ of } \{1, 2, \ldots, N - 1\} \]

Some of the assumptions made in this model are that

1. A customer is satisfied whenever he is serviced.

2. The demand at each node is at most equal to the capacity of the vehicle.

3. Each customer is serviced by one and only one vehicle.

More complicated models have been proposed in the literature, a mixed ILP heterogeneous fleet problem formulation was given by Garvin [GGJS57] in which multiple vehicles may be needed to satisfy a customer. Foster and Ryan [FR76] developed an elaborate formulation which is quite general and they based their approach on
the zero-one ILP formulation of Balinski and Quandt [BQ86] for the vehicle routing problem. Some of these constraints are rather restrictive, for example a customer’s demand may need to be serviced by multiple trucks/vehicles instead of one, and the customer may not be satisfied when he is serviced. Relaxing these constraints makes the problem more unwieldy and not amenable to a clean ILP formulation. Since the approach taken in this thesis is to generate and test several solutions these conditions have been relaxed in the implementation described later in this chapter.

5.2.2 Multi-Depot Vehicle Routing

The ILP formulation of single depot VRP can be modified to incorporate multiple depots by making minor changes in the earlier formulations. All assumptions remain the same, the vehicles can start from any depot, if the nodes

\[(N + 1, \ldots, N + M)\]

represent the M depots the new formulation would be

The problem is to

\[
\min Z = \sum_{i=1}^{N+M} \sum_{j=1}^{N+M} \sum_{k=1}^V c_{ij} x_{ijk}
\]

Each customer is served by exactly one vehicle. This is expressed by constraints 5.12 and 5.13. 5.12 states that only one vehicle enters any given node \(j\), while 5.13 ensures that only one pair \(x_{ij}\) exists such that a vehicle travels from \(i\) to \(j\).

\[
\sum_{i=1}^{N+M} \sum_{k=1}^V x_{ijk} = 1 \text{ for } j = 1, 2, \ldots, N \tag{5.12}
\]

\[
\sum_{j=1}^{N+M} \sum_{k=1}^V x_{ijk} = 1 \text{ for } i = 1, 2, \ldots, N \tag{5.13}
\]

The route continuity is expressed by constraint 5.14, this states that the number of vehicles allocated to a route going into node \(h\) is exactly equal to the number of vehicles leaving node \(h\). The capacity constraint of a vehicle \(P_k\) in 5.15 dictates the
maximum amount of material it can supply to all the nodes occurring in a route through which $P_k$ travels.

\[ \sum_{i=1}^{N+M} x_{ikk} - \sum_{j=1}^{N+M} x_{hjk} = 0 \quad \text{for } k = 1, 2, \cdots, V, \quad h = 1, 2, \cdots, N + M \]  

\[ \sum_{i=1}^{N+M} Q_i \sum_{j=1}^{N+M} x_{ijk} \leq P_k \quad \text{for } k = 1, 2, \cdots, V \]  

(5.14)  

(5.15)

The total route constraint which limits the cost incurred by any given vehicle is given by 5.16

\[ \sum_{i=1}^{N+M} \sum_{j=1}^{N+M} c_{ij} x_{ijk} \leq T_k \quad \text{for } k = 1, 2, \cdots, V \]  

(5.16)

The inequations 5.17, 5.18 make sure that the vehicle availability is not exceeded. The equations 5.20, 5.22 make sure that subtour assignments do not satisfy the previous constraints.

\[ \sum_{i=N+1}^{N+M} \sum_{j=1}^{N} x_{ijk} \leq 1 \quad \text{for } k = 1, 2, \cdots, V \]  

(5.17)

\[ \sum_{j=N+1}^{N+M} \sum_{i=1}^{N} x_{ijk} \leq 1 \quad \text{for } k = 1, 2, \cdots, V \]  

(5.18)

\[ x_{ijk} = 0 \text{ or } 1 \quad \text{for all } i, j, k \]  

(5.19)

\[ x_{ijk} \in S \]  

(5.20)

\[ S = \left\{ (x_{ijk}) : \sum_{i \in B} \sum_{j \in B} x_{ijk} \geq |B| - 1 \text{ for every nonempty subset } B \right\} \]  

(5.21)

(5.22)

It can be seen from the formulations presented that, solving an ILP program to yield a solution is extremely difficult because of the large number of variables. In fact Dantzig and Ramser solved the problem [DR59] by approximating the formulation as a linear programming problem, and the method of solution starts from the basic
idea to synthesize the solution in a number of *stages of aggregation*, in which sub optimizations are carried out on pairs of points or groups.

In the problem they solved, the assumption was that every station’s demand is satisfied by exactly one delivery. Dantzig and Ramser note that by relaxing the condition that demand must be satisfied in full, it may be possible to reduce total truck miles still further, thus it may be possible to over or under deliver up to a fixed percentage based on demand. In the implementation of the transportation cluster done in the current chapter and in the paper [SS92] we allow multideliveries to different destinations.

### 5.3 Ocean Tanker Scheduling

Another very commonly occurring transportation scheduling problem, is the ocean transportation scheduling problem. The large variety of commodities, trades and operating environments in ocean shipping results in a wide spectrum of ship scheduling problems. Ships cost tens of thousand dollars daily, and proper routing/scheduling of a fleet can result in large monetary savings. Ronen [Ron83] provides a comprehensive review of ship routing and scheduling models and problems.

Medium-term scheduling of ships for strategic planning has drawn wide attention in literature Appelgren [App71], Mckay and Hartley [MH71], Laderman, Gleiberman and Egan [LGE66], Olson Sorenson and Sullivan [OSS69] have all done investigation in this area. Levy, Lvov and Lovestsky [LLL77] calculated that the probability that a ship will actually carry out its quarterly schedule is only 0.3 because weather conditions, mechanical problems, port congestion, and labor problems often delay ships. A general summary of the problem structure, assumptions and constraints are provided.

The basic transportation operation:

1. The ships transports bulk or semibulk commodity.
2. All the ships can start from different origins.
3. The ships have multiple destinations during a single voyage.
4. The ships do not try to bring back any return cargo.

5. They have large shipping capacities. Generally each ship is able to service more than one port.

6. There is a limit on the number of unloading ports on a vessel.

The Resources:

1. The ships are transportation resources which deliver commodities to the ports. These have various capacities, speeds, terms of employment and cost of operation functions.

2. A ship may unload in several destination ports.

3. The ship may not return to the starting point at the end of a voyage.

4. There are contract limits on port of call.

The Ports:

1. A destination port may be supplied by several ships.

2. Known unloading rate at each port.

3. Ship draft and length limitations.

The Shipments:

1. Specified by their sizes and destinations.

2. No transhipments allowed

The Costs:

1. Steaming time of ships.

2. Ship’s time at unloading ports.

3. Unit shipping charges.

4. Demurrage

5. Port entry charges for unloading ports
The traditional way to schedule ships which is the industry "rule of thumb" is to send the larger ships to the farther destinations. For example if three ships are available, and the first can carry 10,000 tons, the second 8000 tons and the third 5000, the scheduler will try to load the ship with the cargo to the farthest destination and, if that cargo does not fill the ship, then he may add its cargo to other destinations which are farthest possible in the same direction. In the current implementation of transportation schedulers done in this thesis the user can try to experiment with the most suitable heuristic and can also devise his/her own heuristic to see what heuristic produces the best result. [SS92] also shows how specific knowledge about the current problem domain can help produce better schedules.

The large combinatorial size of the ship scheduling problem necessitates the use of solution algorithms that do not ensure an optimal solution. In the paper Short-Term scheduling of vessels for shipping bulk or semi-bulk commodities originating in a single area, David Ronen [Ron86] considered three algorithm, the first one is an exact algorithm, the second one is a Biased Random Generator Algorithm while the third one is a heuristic single step cost minimization algorithm. The algorithms are discussed in fair detail in his paper, the exact algorithm tries to solve the scheduling problem algorithm by solving many transportation problems (an operations research formulation not to be confused with transportation schedulers) followed by route generation. The biased random generator of schedule is similar to the "generate and test" scheduling algorithm used here, and the heuristic single stop cost minimization algorithm is slightly involved and appears in appendix A3 of [Ron86]. Ronen concluded that the heuristic and biased random algorithms generated schedules cheaper than the industry rule of thumb - which is to dispatch ships first to the farthest location with maximum demand.

The "generate and test" algorithm used in this chapter can be compared with the biased random schedule generator algorithm of Ronen. We believe that the algorithm implemented is more flexible than Ronen's algorithms because it allows for many task orderings based on the experience of the scheduling person, whereas Ronen's algorithm select ships and unloading ports at random.
5.4 A Tanker Scheduler Implementation

A brief introduction to the area of transportation scheduler was presented in the previous sections. In the current section the exact implementation of a tanker scheduler developed under the generic framework is presented. The current tanker scheduling problem implementation involves delivering crude oil to different sites using a fleet consisting of about a dozen tankers. Each of the tanker vessels can be assigned to a voyage and can serve single or multiple ports on a route. The horizon of planning in the present problem is short term i.e. within 30 days, planning is done again for the next month based on resource position and availability at the end of the month. A provision also exists for corrective planning in the system.

The tanker scheduler tries to obtain a good solution, subject to feasibility rules with the overall objective of meeting all the demands for crude oil and obtaining a good low overall cost per barrel for oil delivered to different sites.

The following technical terms used often, need a brief explanation:

*Single Delivery:* While undertaking a single delivery voyage, a tanker discharges its entire load at a single customer location.

*MultiPorting:* The tanker delivers oil to multiple ports in a voyage, these recipient ports lie in its route.

*Lightering:* In this mode of delivery, a tanker “lighters” part of its oil into a lightering vessel. This lightering vessel which collects oil from another ship makes a delivery to a port which is its home base.

5.4.1 Tasks: Specific Oil Delivery Requirements

The specific oil delivery requirement is provided as an input to the program, in the format presented in Table 5.1. The format requires data about the port of delivery, the service zone, minimum and maximum quantity of oil required and the time window between which the delivery needs to be made. An intermediate quantity depending on the resource and route allocation is supplied.
<table>
<thead>
<tr>
<th>Port</th>
<th>Service Zone</th>
<th>$Q_{\text{low}}^{p}$ ($10^6$ brls)</th>
<th>$Q_{\text{high}}^{p}$ ($10^6$ brls)</th>
<th>W1</th>
<th>W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>a</td>
<td>40</td>
<td>50</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2.</td>
<td>b</td>
<td>22</td>
<td>44</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>3.</td>
<td>c</td>
<td>30</td>
<td>60</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>d</td>
<td>45</td>
<td>50</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>5.</td>
<td>e</td>
<td>35</td>
<td>55</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>6.</td>
<td>f</td>
<td>20</td>
<td>30</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>7.</td>
<td>g</td>
<td>25</td>
<td>40</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>8.</td>
<td>h</td>
<td>30</td>
<td>35</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>9.</td>
<td>i</td>
<td>25</td>
<td>35</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 5.1: Sample Task Requirement for a Tanker Scheduling Problem

<table>
<thead>
<tr>
<th>RcsId</th>
<th>Size</th>
<th>Capacity</th>
<th>W1</th>
<th>W2</th>
<th>Loading Cns</th>
</tr>
</thead>
<tbody>
<tr>
<td>tk1</td>
<td>large</td>
<td>100</td>
<td>5</td>
<td>100</td>
<td>none</td>
</tr>
<tr>
<td>tk2</td>
<td>large</td>
<td>100</td>
<td>7</td>
<td>100</td>
<td>none</td>
</tr>
<tr>
<td>tk3</td>
<td>large</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>none</td>
</tr>
<tr>
<td>tk4</td>
<td>medium</td>
<td>60</td>
<td>0</td>
<td>100</td>
<td>none</td>
</tr>
<tr>
<td>tk5</td>
<td>medium</td>
<td>60</td>
<td>0</td>
<td>100</td>
<td>none</td>
</tr>
<tr>
<td>tk6</td>
<td>small</td>
<td>40</td>
<td>0</td>
<td>30</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 5.2: Sample Resources for a Tanker Scheduling Problem

5.4.2 Resources

The resource set $\mathcal{V}$ in the system consists of three kind of tanker vessels: large, medium and small, denoted by the sets $\mathcal{V}^L, \mathcal{V}^M, \mathcal{V}^S$. Large, medium and small tankers can carry enough oil to meet the demand of 3, 2 and 1 ports respectively.

5.4.3 Connectivity Matrix

The distance between ports is supplied by a distance matrix $D$ where each entry $D_{ij}$ is the distance between ports $i$ and $j$. An entry of $u$ for $D_{ij}$ indicates that no route generated should include a direct path from port $i$ to $j$. For example the entries between $i$ and $h$ and between $h$ and $d$ are marked $u$ which indicates that any possible route generated should not include the arcs $i - h$ or $h - d$. 
5.4.4 Constraints

Allocation of resources to tasks is subject to resource constraints and routing constraints, and temporal constraints.

Routing constraints dictate the routes which are taken by various tankers to deliver oil to ports. The two routing constraints in the system are:

Route Length: Restrict the length of routes generated to 3, 2 or 1 for each tanker vessel $V_j \in V^L, V^M, V^S$ respectively.

Unassignable Route Segments: Decrease the number of routes feasible i.e. if a route segment $P_i - P_j$ between ports $P_i$ and $P_j$ is not assignable then no route of length 2 or 3 generated contains the mentioned route leg.

Resource Constraints can be divided into two categories: capacity constraints and infeasible resource assignment constraints. Capacity constraints constrain the resources to deliver oil only to a fixed number of ports as the capacity of the tankers is fixed. Infeasible resource assignments prevent certain tanker vessels from being assigned to serve specific ports.
Temporal Constraints pertain to the times between which the task needs to be completed. The *delivery time window* temporal constraint specifies that for a task $i$ at port $p - T_{ip}$ needs that the delivery be completed between a time window $W^1_i$ and $W^2_i$. Sometimes the temporal constraint and physical location of the destination may conflict, for example, a port at a distance of 5 time units may want a delivery between days 15 and 20, while a port at a distance of 15 days may want a delivery between days 5 and 10. If it becomes very hard to generate schedules within very narrow window ranges, then the window for certain tasks for example window $[W^1_i - W^2_i]$ of task $T_j$ is broadened. Clever routing methods take care of temporal constraints as far as possible.

### 5.4.5 Costs

The costs in the system are made up of *fuel costs*, *idle ship costs* and *miscellaneous costs* which include port entry charges for unloading ports and cost of ships’ time at the unloading port. Fuel costs depend on the type of tanker and actual distance traveled to the destination port, while idle costs are incurred irrespective of delivery and include daily cost of power for auxiliary systems and crew.

To simplify the cost structure we take $c^1_{pu}$ as the unit shipping cost to port $p$ on vessel $v$ and $c^2_{vr}$ is the cost of vessel $v$ taking route $r$. In the present implementation, the relative cost of traveling between destinations $(i,j)$ are assumed to be 2,4,6 per unit distance traveled for small, medium and large vessels respectively. The relative unit shipping cost of a commodity is assumed to be 1,2,3 for the respective vessels.

### 5.4.6 Optimization Formulation for Tanker Scheduler

The transportation problem can be formulated as the following *nonlinear mixed binary program*. The problem is difficult to solve by feeding it to any known constrained optimizer due to the large number of variables involved and also its non-linearity. The variables used in this formulation are first described:
\(Q_p\) is quantity delivered to \(p\)
\(Q_{low}^p\) is min delivery required at \(p\)
\(C_v\) is capacity of vessel \(v\)
\(X, Y\) are decision variables
\(c_{pu}^1\) is unit cost of delivery to
port \(p\) by vessel \(v\)
\(c_{vr}^2\) is cost incurred in route \(r\)
by vessel \(v\).
\(\mathcal{R}_v\) is feasible routes for \(v\).

\[
X_{pvr} = \begin{cases} 
1 & \text{if vessel } v \text{ enters port } p \text{ in route } r \\
0 & \text{otherwise}
\end{cases} \quad Y_{vr} = \begin{cases} 
1 & \text{if vessel } v \text{ takes route } r \\
0 & \text{otherwise}
\end{cases}
\]

The constrained optimization problem is:

\[
\text{Minimize} \left\{ \sum_{v \in V} \sum_{p \in P} \sum_{r \in R_v} c_{pu}^1 Q_p X_{pvr} + \sum_{v \in V} \sum_{r \in R_v} c_{vr}^2 Y_{vr} \right\}
\]

subject to:

\[
\sum_{p \in P} \sum_{r \in R_v} Q_p X_{pvr} \leq C_v \quad v \in V \tag{5.23}
\]

\[
Q_p \geq Q_{low}^p \quad p \in P \tag{5.24}
\]

\[
\sum_{r \in R_v} Y_{vr} = 1 \quad v \in V \tag{5.25}
\]

Constraint 5.23 ensures that the total deliveries are within capacities of the vessels, constraint 5.24 ensures that only one route is taken by a tanker, constraint 5.25 ensures the delivery of correct quantities of oil.

### 5.5 Grammar for Transportation Cluster

In this section the grammar for recognizing the different elements of the transportation cluster is presented. Prolog Definite Clause Grammar rules are used to parse the input to the scheduling problem. The top level rule is scheduler/9 which collects all the required information about the current scheduler, the information fields
which define the scheduling cluster are, dataset information - which keeps track of
the problem which is being solved, the tasks in the system, resources in the system,
the connectivity matrix which indicates terminal to destination and inter destination
distances, heuristics used for scheduling, and the service strategy which dictates how
deliveries can be made during the schedule. Other important parameters include
constraints in the system, cost of distribution and maximum number of iterations
after which the best schedule is selected. phrase/2 is the Prolog predicate which
parses the input stream and collects scheduler specific information.

5.5.1 Task Specification for a Transportation Scheduler
The main fields of a task object are the task identifier TaskId, the delivery ad-
dress Address, and the zonal information zone which identifies the region in which
the particular delivery address exists. The quantity of delivery which needs to be
made lies within a range qlow and qhigh, and the delivery window i.e. the times
between which the delivery has to be made is specified in the variable w1 and w2.
 serv_stations contains the list of service stations which can deliver to the station
in question. The extra_fields is a list of additional fields which are new attributes
which any user can define for any task in the current scheduling instance. The extra
fields are mainly used during customization of a scheduler and contain any fields
specific to the tasks in the current scheduler instance.

%——DCG For Recognizing a Task ———

\[
\text{task}(\text{task}([\text{task}._\text{id}::\text{TaskId}, \text{addr}::\text{Address}, \text{zone}::\text{Zone,}
\text{serv}._\text{stations}::\text{Stations,}
\text{qlow}::\text{QLow}, \text{qhigh}::\text{QHigh}, \text{w1}::\text{W1}, \text{w2}::\text{W2}|\text{ExtraFields}]))
\rightarrow
\text{taskId}(\text{TaskId}), [';'],
\text{address}(\text{Address}), [';'],
\text{zone}(\text{Zone}),
\text{service}(\text{Stations}), [';'],
\text{quantity}(\text{QLow}, \text{QHigh}), [';'],
\text{delivery\_window}(\text{W1}, \text{W2}), [';'],
#
\]
extra_fields(ExtraFields),
#
taskId(TestId)
    → ['task_id', ':', 'TestId', '{atomic(TestId)}].
address(Address)
    → ['address', ':'], [Address].
zone(Zone)
    → ['zone', ':'], [Zone], [':'].
zone([]) → [].
service(Stations)
    → ['service_stations', ':', '[]'], service1(Stations).
service1(Stations)
    → [Station, ''], service1(RestStations), {Stations = [Station|RestStations]}. service1([Station])
    → [Station, '']
quantity(qlow::QLow, qhigh::QHigh)
    → ['required_quantity', ':'], [QLow], [QHigh],
        {integer(QLow), integer(QHigh)}. delivery_window(w1::Win1, w2::Win2)
    → ['delivery_window', ':'], [Win1], [Win2], {integer(Win1), integer(Win2)}. delivery_window([]).
% ——— DCG for collecting all tasks information ——— tasks(Tasks)
    → [tasks, ''], tasks1(Tasks).

tasks1(Tasks)
    → task(Task), tasks1(RestTasks), {Tasks = [Task|RestTasks]}. tasks1([]) → []

An example task translation is shown here. The task is translated into an inter-
mediate form, the first task has a task identifier task_id task1, an address a, is in
zone 1, can be served by a list of service station serv_stations {t1}. 
<table>
<thead>
<tr>
<th>Task_Id</th>
<th>Address</th>
<th>Zone</th>
<th>service_stations</th>
<th>required_quantity</th>
<th>Delivery_Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>task1;</td>
<td>a;</td>
<td>1;</td>
<td>[t1];</td>
<td>40 50;</td>
<td>1 30;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>priority 3;</td>
<td></td>
<td>order 1;</td>
</tr>
<tr>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task_Id</th>
<th>Address</th>
<th>Zone</th>
<th>service_stations</th>
<th>required_quantity</th>
<th>Delivery_Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>task2;</td>
<td>b;</td>
<td>1;</td>
<td>[t1];</td>
<td>22 44;</td>
<td>1 30;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>priority 2;</td>
<td></td>
<td>order 2;</td>
</tr>
<tr>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Task Field Description for Transportation Scheduler

Task `task1` requires delivery to be made between $40 \times 10^4$ and $50 \times 10^4$ barrels of oil which are parameters `qlow` and `qhigh`, the delivery is to be completed between days 1 and 30 which are parameters `w1` and `w2`. The scheduling person can also define a few extra fields in Figure 5.4. The two user defined fields defined here are `priority` and `order`, a priority may be added to any task and the task selection heuristics may be asked to dispatch tasks based on the priority of tasks. The order field may be used when the scheduling person wants tasks to be dispatched in a fixed order as indicated by the order field. Priority of `task1` is 3 in this example and the order is 1, both priority and order may not be used at the same time to determine the order of dispatch by a heuristic. The task input is translated to a Prolog intermediate form:

```prolog
  task([task_id:task1, addr:a, zone:1, serv_stations:[t1], qlow:40, qhigh:50, w1:1, w2:30, user_defined:priority:3, user_defined:order:1])
  task([task_id:task2, addr:b, zone:1, serv_stations:[t1], qlow:22, qhigh:44, w1:1, w2:30, user_defined:priority:2, user_defined:order:2]).
```

5.5.2 Transportation Scheduler Resource Specification

The main fields for a resource used for allocation in a transportation scheduler are a resource identifier `rcs_id`, size of the transporting vessel `size`, the capacity of the vessel `cap`. The time window during which it is available are given by the values of `w1` and `w2`, the compartments of the vessels is given by the `compartments` field.
Loading constraints can be specified in loading, the maximum number of deliveries allowed is given by the value of n\_dlvrys.

%—DCG For Recognizing a Resource——

resource(rcs([rcs\_id::TankerId, size::Size, cap::Capacity, w1::W1, w2::W2,
               n\_dlvrys::N\_Dlvrys[ExtraFields]]))
→ rcs\_id(TankerId),[;'],
   size(Size),[;'],
   capacity(Capacity),[;'],
   avail\_window(W1,W2),[;'],
   n\_dlvrys(N\_Dlvrys),[;'],
   #,
   extra\_fields(ExtraFields),
   #.

rcs\_id(TankerId)
→ ['rcs\_id',';'], [TankerId].

size(Size)
→ ['size',';'],[Size].

capacity(Capacity)
→ ['capacity',';'],[Capacity].

avail\_window(W1,W2)
→ ['avail\_win',';'],[W1,W2],{integer(W1),integer(W2)}.

avail\_window([],[]) → [].

% ——— DCG for collecting all resource information ———

resources(Resources)
→ [resources],[;:],resources1(Resources).

resources1(Resources)
→ resource(Rcs),resources1(RestRcs),{Resources=[Rcs|RestRcs]}.

resources1([]) → [].

The following example shows a sample resource specification, one of the resources shown has a resource identifier tk1, the size of the vessel is large, the capacity of the vessel is \(100 \times 10^4\), and the vessel is available for a voyage starting from day 5 to day 100, the vessel does not have any special loading constraint. Tanker tk1 has an extra field order whose value is 1, this field can be used by an ordering heuristic.

<table>
<thead>
<tr>
<th>resources ::</th>
<th>resources ::</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rcs.Id : tk1;</td>
<td>Rcs.Id : tk2;</td>
</tr>
<tr>
<td>size : large;</td>
<td>size : large;</td>
</tr>
<tr>
<td>Capacity : 100;</td>
<td>Capacity : 100;</td>
</tr>
<tr>
<td>Avail.Win : 5 100;</td>
<td>Avail.Win : 7 100;</td>
</tr>
<tr>
<td>n_dlrvys : 0;</td>
<td>n_dlrvys : 0;</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>order : 1;</td>
<td>order : 2;</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

Table 5.5: Resource Field Description for Transportation Scheduler

### 5.5.3 Transportation Scheduler Constraint Specification

Constraints in the system prevent the assignment of certain vessels to ports, the two common kinds of constraints in the system are delivery constraints and port constraints.

```
%— Recognizing Constraints in the system —

constraints(Constraints)
  → [constraints,'::'],port.constraints(PortConstraints),
     delivery.constraints(DeliveryConstraints),
     min.quantity.transportable(MinQuantity),
     {Constraints=[port.cns::PortConstraints,delivery.cns::
       DeliveryConstraints,min.quant::MinQuantity]}.

constraints([]) → [].

port.constraints(Constraints)
  → ['forbidden.deasts','::'],[Transporter,'[]],
```
The `port_constraints` prevent the allocation of some resource to specific tasks. In the example presented in Table 5.6 tanker tk1 cannot be allocated to deliver to port b, similarly tanker tk2 and tanker tk3 cannot be allocated to ports d and e respectively. The `delivery_constraints` decide the number of destinations tankers of different categories can deliver to. Tankers whose size is small can deliver to only 2 ports, medium size tankers can deliver to 3 destinations while large tankers can deliver to 4 destinations on the same round. This constraint affects routing generation. It is similar to constraints in both the job shop and time tabling cluster, and can be related to the `capacity constraint` of resources. It is similar to putting a limit on the number of sessions which can be scheduled in a venue in the case of a timetabling scheduler, as well as the maximum number of tasks a particular operator can be assigned to in the case of shop floor schedulers.

### 5.5.4 Transportation Scheduler Parameters Specification

The costs in the system are made up of `fuel costs`, `idle ship costs` and `miscellaneous costs` which include port entry charges for unloading ports and cost of ships' time.
at the unloading port. Fuel costs depend on the type of tanker and actual distance traveled to the destination port, while idle costs are incurred irrespective of delivery and include daily cost of power for auxiliary systems and crew.

\[
\text{min\_quantity\_transportable}(\text{MinQuant}) \\
\rightarrow \text{['min\_quantity\_transportable',':'},\text{MinQuant},';'].
\]
\[
\text{costs}(\text{Costs}) \\
\rightarrow \text{[costs,:''],maintenance\_cost(\text{MaintenanceCost}),} \\
\text{cost\_per\_distance(\text{TravelingCost}), merit\_cost(\text{MeritCost}),} \\
\{\text{Costs=[maint\_Cst::MaintenanceCost, trvl\_Cst::TravelingCost,} \\
\text{merit\_Cst::MeritCost]}.}
\]
\[
\text{maintenance\_cost(\text{Maintenance\_Cost})} \\
\rightarrow \text{['maintenance\_cost',':'],[\text{SizeCategory1},[\text{Cost1},';']}, \\
\text{maintenance\_cost(\text{RestCosts}),} \\
\{\text{Maintenance\_Cost=[(\text{SizeCategory1,Cost1})|\text{RestCosts}]}\}.
\]
\[
\text{maintenance\_cost([]} \rightarrow \text{[]}. \\
\text{cost\_per\_distance(\text{Cost\_per\_distance})} \\
\rightarrow \text{['cost\_per\_distance',':'],[\text{SizeCategory1},[\text{Cost1},';']}, \\
\text{cost\_per\_distance(\text{RestCosts}),} \\
\{\text{Cost\_per\_distance=[(\text{SizeCategory1,Cost1})|\text{RestCosts}]}\}.
\]
\[
\text{cost\_per\_distance([]} \rightarrow \text{[]}. \\
\text{merit\_cost(\text{MeritCost})} \rightarrow \text{['merit\_cost',':'],[\text{MeritCost},';']}. \\
\text{max\_iterations(\text{MaxIterations})} \rightarrow \text{['max\_iterations',':'],[\text{MaxIterations},';']}.\]
In the actual implementation we have simplified the cost structure, each vessel has a maintenance cost, for example in Table 5.7 the small ship has a daily maintenance cost of 10 units, the medium ship has a daily maintenance cost of 20 units while the large ship has a daily maintenance cost of 30 units. These include the cost of crew and other supplies for the vessels. The fuel costs are taken into account by cost_per_distance which is proportional to the size of the ship.

| costs:  |
|---|---|
| maintenance_cost : small  | 10; |
| maintenance_cost : medium | 15; |
| maintenance_cost : large  | 20; |
| cost_per_distance : small  | 10; |
| cost_per_distance : large  | 20; |
| cost_per_distance : medium | 30; |

max_iterations : 30.

Table 5.7: Simplified Cost Information for a Transportation Scheduler

5.5.5 Extra Fields

The specification of fields in each of the scheduler clusters is extensible because there is a provision for including extra fields after specifying the standard fields both in tasks and resources. The extra fields are enclosed within the '‐' character, the DCG grammar for handling extra_fields has been provided later on in this subsection. Each of extra fields are stored as user_defined::Fields. For example the task extra fields priority:3, order:1 are stored as user_defined::priority:3 and user_defined::order:1. Extra fields also help in defining new schedule heuristics, where the precise order of task selection and resource ordering can be determined by defining pseudo fields with the order of scheduling depending on these extra fields.
5.6 Transportation Schedule Heuristics

Each scheduler is guided by schedule heuristics which basically determine two things, the next task to be scheduled and the resource to be allocated to the scheduled task. `choose_task` and `choose_update_resources` need more information about making their choice. The heuristics selector provides control information to the basic schedule cycle. An example specification of a schedule heuristic is shown in Figure 5-2.

5.6.1 Pre-Defined Heuristics

Certain heuristics are pre-defined in the system for both order of task selection and resource allocation. Examples of some common pre-defined task selection heuristics are `Largest Delivery First` and `Farthest Task First` in the case of transportation schedulers. Examples of pre-defined resources selection for the Transportation schedulers are `Highest Resource Capacity First` and `Most Constrained Vessel First`. The pre-defined heuristics provide the user a convenient way of using the scheduler to run simulations to see scheduling results. Some of the results obtained when the heuristics were applied to different datasets have been presented in Section 5.8.

The order of task allocation and resource allocation is very important because they determine the number of iterations in which good schedules can be reached. Brief results provided in Section 5.8 indicate the importance of task and resource ordering heuristics.

Tasks are ordered by task ordering heuristics. Task selection heuristics enable the system to get good feasible solutions in fewer iterations. One of the task ordering
tried was random while other orderings are obtained by using the following task ordering heuristics:

**Highest demand first**: Tasks with more demand are scheduled first. Each of the task has a `qlow` field which indicates the minimum amount of crude which needs to be supplied to the current destination. In this heuristic all tasks are first sorted using `qlow` being the primary key. Olson Sorenson and Sullivan [OSS69] have also used the same strategy as one possible heuristic for dispatching oil tankers to medium range destinations. Tasks with higher demands are considered as more difficult tasks, and consequently must be accomplished first. It is always possible to squeeze in tasks with lower demand later, the only problem being the physical proximity of the later tasks to the previous ones.

**Farthest tasks first**: Tasks which are at a greater distance from the source are scheduled first. Distance from the terminal is also a good measure of the difficulty of the task, the more farther a task is the more time it ties up the resource which has been dispatched to deliver material to it. For example in Figure 5-1, routing different tankers first to ports c, a and h may provide good feasible solutions rather than satisfying the demands of ports e, f and d. This heuristic is also called the industry rule of thumb and has been cited as quite widely used in a survey paper by Ronen [Ron86].

**Clustered Proximal tasks with highest demand first**: Ports which are close to one another are arranged in a cluster which is a group of two or three ports, and it is attempted to service them by using the same resource. This is shown in Figure 5-1 where tasks which are close are routed together from the outset. The transportation scheduling algorithm presented in Figure 5-3 dispatches tasks one after the other, and uses the same resource until the resource capacity is exceeded after which a new resource is selected. It is thus better to have tasks which are proximal to one another to be dispatched consecutively rather than tasks which lie far away from each other. As an example tasks \{b, c, d\}; \{e, f\} are ordered in sequence so that the same vessel could handle these tasks clusters. The cost savings comes from the fact that the transporting
vessel has to travel less distance for delivery. If tasks were dispatched \{c, h, a\} in that order then the routing cost will be very high. This heuristic may not always work because the same resource may not have sufficient capacity to service all proximal ports.

Figure 5-1: Routing Clustered Proximal Tasks Together

A resource from \( V \) needs to be selected for each task \( T_i \). Similar to the heuristics used for task selection, several resource selection heuristics were tried in order to get a good match between the task \( T_i \) and the resource \( V_i \) assigned to it. Two heuristics implemented are:

**Higher Resource Capacity First**: Resources with larger capacity are considered first, i.e. larger vessels are tried before smaller ones. Ties between vessels
of the same class are broken arbitrarily.

Most Constrained Vessel First: Resources with more constraints in terms of which ports can be visited are allocated before other less constrained vessels. Ties are broken by higher resource capacity. This is similar to the task scheduling heuristic schedule "most conflicting sessions first", in the case of timetabling problems. This heuristic is implemented in the belief that the most constrained vessels are bottleneck resources and should be dispatched first.

```
| Task_Heuristics: user_defined; |
| Heuristic_Id : 1; |
| Operations |
| [ |
| Operation: sort |
| on Primary (user_defined.order) asc; |
| ]; |

| Task_Heuristics: user_defined; |
| Heuristic_Id : 2; |
| Operations |
| [ |
| Operation: select; |
| Conditions: [(user_defined.priority), ≥, 3]; |
| Operation: sort |
| on Primary (standard.qhigh) asc; |
| ]; |
| [Operation: choose_remaining]; |

| Task_Heuristics: pre_defined; |
| Heuristic_Id : 3; |
| Operation: highest_demand_first; |
| Task_Hs_Execution_Order: [2,3,1]; |

| Resource_Heuristics: user_defined; |
| Heuristic_Id : 2; |
| Operations |
| [ |
| Operation: sort |
| on Primary (user_defined.order) asc; |
| ]; |

| Resource_Heuristics: user_defined; |
| Heuristic_Id : 1; |
| Operations |
| [ |
| Operation: select; |
| Conditions: [(standard.capacity), ≥, 45]; |
| Operation: sort |
| on Primary (standard.w1) asc; |
| ]; |
| [Operation: choose_remaining]; |

| Resource_Hs_Execution_Order: [2,1]; |

Figure 5-2: Example Heuristic Specification for a Transportation Scheduler

5.6.2 User-Defined Heuristics

Sample user-defined heuristics have been shown in Figure 5-2. The first task heuristic sorts all the given tasks based on an extra-field order (extra-field has been described previously in Subsection 5.5.5). The user can precisely control the order of task selection by defining this extra-field and numbering the extra-fields 1,2,...
where the order signifies the order in which the tasks will be scheduled. The exact ordering of tasks is achieved by sorting the tasks, using order as the primary key. The second user-defined heuristic selects tasks which have a priority greater than 3, and further schedules the task with the highest "maximum demand" first (qhigh is the maximum demand at a port). Remaining tasks with priorities less than 3 are scheduled later as specified by the instruction Operation: choose_remaining.

5.6.3 Heuristic Execution

Task and resource heuristic execution order are specified in the input. In the present example as indicated in Figure 5-2 the heuristic orders of task and resources are specified as Task.Hs.Execution_Order : [2, 3, 1] and Resource.Hs.Execution.Order : [2, 1] respectively. During schedule execution the first pair of heuristics executed are <2,2> for task and resource i.e. user-defined task heuristics 2 and user-defined resource heuristic 2, and similarly other elements from the cartesian product of Task.Hs.Execution.Order and Resource.Hs.Execution.Order are tried out in order. The best schedule is reported from the given sequence.

We feel that by providing a mechanism for customizing the control of the scheduler, a flexible scheduler has been developed, in contrast to the schedulers existing in industry.

5.7 Generate and Test Algorithm

Allocation of tankers to different routes, serving ports is achieved by the transportation scheduling algorithm presented in figure 5-3. The main steps in the algorithm are Task Selection, Resource Selection, Resource Allocation and updating of Task, Resource List. Once the Resource Allocation is completed Route Assignments are done for each resource serving destinations. The merit of a schedule is determined by Cost Calculations and schedules are improved by backtracking.

Once resource \( V_i \) is allocated to task \( T_i \) at destination \( P_i \), the available capacity of the resource is updated. The demand at any port can be set to a quantity greater than the quantity \( Q_{low} \) for the port. At the calculations stage, step 5 in the algorithm, adjustments are made to determine the exact quantity of oil delivered
Figure 5-3: Generate and Test Algorithm for a Transportation Scheduler

to each port. The resource remains as an available choice for the next task if it contains at least the MinimumDeliverableQuantity of space. Some of the steps in
Step 5. **Calculations:**

**Delivery Adjustments:** Adjust actual delivery sizes for all $V_i$-[Ps] depending on spare capacity of $V_i$.

**Total Cost:** Calculate routing and delivery costs

$\forall V_i \in \mathcal{V}$ of the schedule.

Compare cost of current schedule and store $S$ if better solution.

if (#SchedulesGenerated $> N$)
go to Step 6.
else
backtrack on choicepoints.

Step 6. **Schedule Output:**

Print best solution $S$ generated.

Stop.

---

**Figure 5-4:** Generate and Test Algorithm for a Transportation Scheduler Continued

```prolog
schedule([], Resources, Allocation, Allocation).

schedule(Tasks, Resources, CurrentAllocation, Allocation) :-
    choose_task(Task, Tasks, RestTasks),
    choose_update_resources(Task, Resources, AssignedRcs, UpdtRcs),
    allocate(Task, AssignedRcs, CurrentAllocation, PartialAllocation),
    schedule(RestTasks, UpdtRcs, PartialAllocation, Allocation).

choose_task(Task, Tasks, RestTasks) :-
    select(Task, Tasks, RestTasks), !.

choose_update_resources(task(Task_F), Resources, AssignedRcs, UpdtRcs) :-
    select(rcs(Rcs_F), Resources, RemRcs),
    access.Field.val([addr::Task_F,[Address]],
    access.Field.val([rcs_id::Rcs_F,[RcsId]],
    can_visit(Address, RcsId),
    assign_resource(task(Task_F), rcs(Rcs_F), AssignedRcs, RemRcs, UpdtRcs).
```

**Figure 5-5:** Sample Tanker Scheduling Predicates

the algorithm are explained in more detail in Subsections 5.7.1, 5.7.2, 5.7.3, 5.7.4.
5.7.1 Prolog Schedule Predicates

Schedule generation is done in two phases, in the first phase DCG rules are used to parse the input and in the second phase the predicate transport_scheduler is called, this predicate is shown in Figure 5-5. transport_scheduler takes Dataset, task heuristic identifier TaskHsId, resource heuristic identifier RcsHsId, pre-ordered tasks OrderedTasks and ordered resources OrderedResources as input and generates a suitable schedule Schedule having the lowest cost Cost within a prescribed set of iterations.

Scheduling proceeds by generating a basic schedule, followed by generation of route and exact material allocation. The schedule predicate implements our generate and test scheduler and is used in all the scheduling clusters, which goes on to show that the philosophy of design of the different schedulers is the same for developing different schedulers. Each of the steps 1,2,3,4 are captured by the schedule/4 predicate which invokes predicates choose_task, choose_update_resource/4, allocate/4. The tasks have already been pre-ordered based on the current task heuristic and choose_task selects the the next task which needs to be scheduled using the select/3 predicate. Once the particular resource is selected choose_update_resource/4 selects the next available resource for dispatch and makes sure that the selected resource can visit the current task location, next the predicate assign_resource is invoked, to complete resource assignment.

5.7.2 Resource Selection:

Sample predicates for resource selection/allocation and updating of resource are shown in Figure 5-6. The first predicate assign_resource/4 checks if the minimum amount of fuel requested qlow can be delivered by the current selected resource. If it can, then the available space in the current resource is updated by the predicate update_resource/4. Sometimes the current resource may not have the ability to fully satisfy the current task requirement. The second assign_resource/2 predicate removes the current resource from the resource list after listing it as a partial supplier to the current task. The oil requirement of the current task is updated, taking into account the amount delivered by the current resource, and the predicate
choose_update_resource is invoked to continue resource selection for the current task. update_resource/4 ensures that the present tanker is shipping at least the minimum quantity of oil which can be shipped, and has not exceeded the upper limit on the number of deliveries it can make. The number of ports to which deliveries have been made is incremented by increment ndlvrys/2 and the fields of the new resource object are updated by assign Fld_val which assigns new values to the n_dlvrys and cap resource fields.

% single resource assigned to the task
assign_resource(task(Task_F),rcs(Rcs_F),[rcs(Rcs_F)],[RemRcs,UpdtRcs]):-
  access_Fld_val([qlow]:Task_F,[Qlow]),
  access_Fld_val([cap]:Rcs_F,[AvSp]),
  AvSp ≥ Qlow,
  update_resource(rcs(Rcs_F),Qlow,RemRcs,UpdtRcs),
  !.

% more than one resource are required for the same task.
assign_resource(task(Task_F),rcs(Rcs_F),[rcs(Rcs_F)],[Rcs2],[RemRcs,UpdtRcs]):-
  access_Fld_val([qlow]:Task_F,[Qlow]),
  access_Fld_val([cap]:Rcs_F,[AvSp]),
  MoreReqd is Qlow-AvSp,
  assign_Fld_val([qlow]:Task_F,[MoreReqd],UpTask_F),
  choose_update_resources(task(UpTask_F),RemRcs,Rcs2,UpdtRcs).

update_resource(rcs(Rcs_F),Qlow,RemRcs,[rcs(UpRcs_F)],[RemRcs]) :-
  min_quantity_transportable(MinQuantityTransportable),
  access_Fld_val([cap,size,n_dlvrys]:Rcs_F,[AvSp,Size,N_Dlvrys]),
  not_exceeded_delivery_visits(Size,N_Dlvrys),
  AvSp ≥ Qlow + MinQuantityTransportable,
  !,
  increment ndlvrys(N_Dlvrys,NewN_Dlvrys),
  RemCapacity is AvSp - Qlow,
  assign_Fld_val([n_dlvrys,cap]:Rcs_F,[NewN_Dlvrys,RemCapacity],UpRcs_F).

Figure 5-6: Resource Allocation and Updating in Prolog
% Current resource has been allocated, one more destination is added.
allocate(task(Task_F),[rcs(Rcs_F)]RestAlloc,CurrentAlloc,UpAllocation):-
access_Fld_val([rcs.id::Rcs_F,[RcsId1]],
access_Fld_val([addr::Task_F,[Address]],
select(alloc(Alloc_F),CurrentAlloc,OtherAllocs),
access_Fld_val([rcs.id::Alloc_F,[RcsId1]], /* check if same */
add_Fld_val([route::Alloc_F,[Address],UpAlloc_F],
allocate(task(Task_F),RestAlloc,[alloc(UpAlloc_F)|OtherAllocs],UpAllocation),
!).

% A new Allocation is made as resource is not previously allocated.
allocate(task(Task_F),[rcs(Rcs_F)]RestRcs,CurrentAlloc,UpAllocation):-
access_Fld_val([addr::Task_F,[Address]],
access_Fld_val([rcs.id,size,w1::Rcs_F,[RcsId,Size,StartT]],
make_object(alloc,[rcs.id::RcsId,size::Size,route::[Address],w1::StartT],AllocObj),
allocate(task(Task_F),RestRcs,[AllocObj|CurrentAlloc],UpAllocation),
!).

% allocate terminates once all resources are exhausted.
allocate(-Task,[],UpdatedAllocation,UpdatedAllocation).

Figure 5-7: Prolog Predicates for Allocation

5.7.3 Allocation and updating:
The allocation done at Step 3 pairs resources with tasks. The different fields in an alloc object are res.id which is the resource identifier, size which is the size of the allocated resource, route which is the actual shortest distance route taken by the tanker vessel, w1 which is the start time of the journey. Three clauses of the allocate/4 predicate are presented in Figure 5-7, the first one adds a task to the destination list of a resource which is already in use, while the second predicate adds the task to the destination list of a resource which has not been allocated to any task. The first predicate checks all allocations currently made by selecting each one of the alloc objects and testing if the resource identifier in the particular object is same as the identifier of the current resource which is being allocated. If the select/3 function fails to produce an alloc object which uses the current resource, the second allocate/4 predicate creates a new object alloc.
The shortest route is generated by obtaining permutations of destinations on route and picking out the permutation which involves the shortest dist.

```prolog
generate_shortest_route([[],[]]).

generate_shortest_route([alloc(Alloc_F1)|Rest1],[alloc(Alloc_F2)|Rest2]):-
    generate_shortest_route1(alloc(Alloc_F1),alloc(Alloc_F2)),
    accessFld_val([rt_dist::Alloc_F2,[RouteDist]]),
    RouteDist ≠ 1000, /* no route possible for the allocation */
    generate_shortest_route(Rest1,Rest2), !.

generate_shortest_route1(alloc(Alloc_F),alloc(NewAlloc_F)):-
    accessFld_val([route::Alloc_F,[Destinations]]),
    setof(Order, permutation(Destinations,Order),DestOrderings),
    get_shortest_dist(DestOrderings,[],1000,BestRoute,ShortestDist),
    assignFld_val([route::Alloc_F,[BestRoute],Alloc_F1]),
    addFlds(Alloc_F1,[rt_dist::ShortestDist],NewAlloc_F), !.

generate_shortest_dist([],BestRoute,ShortestDist,BestRoute,ShortestDist).

generate_shortest_dist([Ordering|RestOrdering],Route1,D1,BestRoute,ShortestDist):-
    get_distance(Ordering,RouteDist),
    choose_better(Route1,D1,Ordering,RouteDist,BetterRoute,SmallerDist),
    generate_shortest_dist(RestOrdering,BetterRoute,SmallerDist,BestRoute,ShortestDist), !.
```

Figure 5-8: Sample Prolog Predicates for Route Generation

### 5.7.4 Routing

Routing occurs at step 4 in the transportation scheduling algorithm presented in Figure 5-3. Routing is done for each resource which may be allocated to a collection of tasks. Since one of the major factors contributing to the cost of the schedule is the actual distance traveled by vessels, the best possible route i.e. the route which has the least total distance traveled is generated for each resource/tasks pairs.

The shortest route for each vessel is generated by obtaining all permutations of the destinations to which deliveries need to be made, and picking up the route which
has the minimum distance. The distance between ports is provided in the connectivity matrix, and routing constraints may dictate that certain segments cannot be present on any route. Some of the Prolog predicates used in route generation are presented in Figure 5-8, generate_shortest_route/2 takes each of the resource/task allocations generated during the schedule and uses generate_shortest_route1/2 to actually generate the shortest route between this resource destination combination. The setof/3 predicate is used to generate the set of all destination ordering. Since the maximum number of destinations which can be on a route is a very low number typically between 1 and 5, the number of routes generated are not too many. get_shortest_distance/5 keeps track of the current shortest route, it starts out by assuming the best route to be a fictitious one having a high distance of 1000. If no routes can be generated then the best route generated still has a distance of 1000 while in all other cases a route with a lower distance is generated.

A new field rt_dist whose value is the minimum distance route is added to each allocation by generate_shortest_route1/2. If the no feasible route can be generated between the destinations the predicate generate_shortest_route/2 fails and the system backtracks to generate new resource/task allocations.

5.8 Results and Discussion

The code for tanker scheduling has been developed based on the algorithm presented in the current chapter. We present the results of two task and resource heuristics each of which have been implemented for task and resource selection and these are

Task Selection: Highest Demand First, and Farthest Task First.

Resource Selection: Most Constrained Resource First, and Largest Capacity First.

For dataset 4 we show the effect of dispatching clustered tasks first as opposed to dispatching highest demand tasks first, or farthest tasks first. The two heuristics possible in conjunction with dispatching clustered tasks together are dispatching
clustered tasks with smaller demand tasks scheduled first, and dispatching clustered tasks with higher demand tasks scheduled first.

A brief description of each dataset is provided together with a discussion of the performance of each heuristic. We have assumed the costs to be constant over all datasets, there is a limit to the number of destinations which big, medium and small vessels can make. For the delivery of each unit quantity to a port above the minimum required quantity, the cost of the schedule is reduced by a merit cost, this is done so that maximum possible delivery of oil is done to all ports. The time constraint for delivery has been relaxed, i.e. there is no window $W^1,W^2$ for each task between which the deliveries have to be made.

## Constraints and Costs

```prolog
max_ports_allowed(big,4).
max_ports_allowed(medium,2).
max_ports_allowed(small,1).

basic_maintenance_cost(big,300).
basic_maintenance_cost(medium,150).
basic_maintenance_cost(small,75).

cost_per_distance(big,75).
cost_per_distance(medium,50).
cost_per_distance(small,25).
min_quantity_transportable(15).
merit_per_extra_quantity_delivery(15).
```

### Dataset 1

<table>
<thead>
<tr>
<th>Number Of Tasks</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Of Resources</td>
<td>5</td>
</tr>
</tbody>
</table>
Number of Constraints = 10.

Dataset1: Tasks

<table>
<thead>
<tr>
<th>Port</th>
<th>( Q_{low}^p ) (10^6 brls)</th>
<th>( Q_{high}^p ) (10^6 brls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PortA</td>
<td>35</td>
<td>55</td>
</tr>
<tr>
<td>2. PortB</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>3. PortC</td>
<td>42</td>
<td>50</td>
</tr>
<tr>
<td>4. PortD</td>
<td>24</td>
<td>40</td>
</tr>
<tr>
<td>5. PortE</td>
<td>46</td>
<td>55</td>
</tr>
</tbody>
</table>

Dataset1: Resources

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Category</th>
<th>MaxCapacity</th>
<th>Earliest Date Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>medium</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>t3</td>
<td>medium</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>t4</td>
<td>medium</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>t5</td>
<td>small</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

Dataset1: Infeasible Resource-Task Allocation Constraints

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Port</th>
<th>Port</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>c</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>e</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Dataset 1: Connectivity matrix

<table>
<thead>
<tr>
<th>Ports</th>
<th>s1</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>0</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>0</td>
<td>u</td>
<td>8</td>
<td>7</td>
<td>u</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>u</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>8</td>
<td>0</td>
<td>u</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>u</td>
<td>0</td>
<td>u</td>
</tr>
<tr>
<td>e</td>
<td>15</td>
<td>u</td>
<td>8</td>
<td>7</td>
<td>u</td>
<td>0</td>
</tr>
</tbody>
</table>

Best Generated schedule

Cost = 16210

Dataset 1: A Generated Schedule with cost 26875 is:

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Size</th>
<th>Route Distance</th>
<th>Start Day</th>
<th>Ports On Route</th>
<th>Arrival (Day)</th>
<th>Min Required</th>
<th>Quantity Delivered</th>
<th>Day Avail for Next Voyage</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>33</td>
<td>0</td>
<td>a</td>
<td>8</td>
<td>35</td>
<td>35</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td>16</td>
<td>42</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>med</td>
<td>24</td>
<td>0</td>
<td>d</td>
<td>12</td>
<td>24</td>
<td>40</td>
<td>24</td>
</tr>
<tr>
<td>t3</td>
<td>med</td>
<td>20</td>
<td>0</td>
<td>c</td>
<td>12</td>
<td>42</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>t4</td>
<td>med</td>
<td>30</td>
<td>0</td>
<td>e</td>
<td>15</td>
<td>46</td>
<td>50</td>
<td>30</td>
</tr>
</tbody>
</table>
Dataset 1 Results:

Dataset 1 consists of 5 tasks, these are deliveries which need to be made to ports A, B, C, D, E and these have to be done using a maximum of 5 tankers t1, t2, t3, t4, t5. The tanker t1 is a big tanker while t2, t3, t4 are medium in size. Resource assignment infeasibilities have been shown with the given data. The best schedule using the given heuristics had a schedule cost of 16210 units, a generated schedule with a cost 26875 has been provided with Dataset 1 information. Figure 5-9 is a plot of the task ordering heuristic Higher Demand Tasks scheduled first along with the resource heuristic which allocates most constrained resources first. The performance of all the four heuristic combinations was not very different for this dataset. The higher demand first, and most constrained resource first heuristic combination provided the optimal solution in about 150 iterations, whereas the heuristic higher demand task scheduled first, and resource heuristic scheduling resources with larger capacity first provided the optimal solution in 180 iterations. Scheduling farther tasks first was marginally better than scheduling higher demand tasks first.
Dataset2

Number Of Tasks = 10
Number Of Resources = 4
Number of Constraints = 18

Dataset2: Tasks

<table>
<thead>
<tr>
<th>Port</th>
<th>$Q_{low}^P (10^6 \text{ brls})$</th>
<th>$Q_{high}^P (10^6 \text{ brls})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Port a</td>
<td>45</td>
<td>55</td>
</tr>
<tr>
<td>2. Port b</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td>3. Port c</td>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>4. Port d</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>5. Port e</td>
<td>35</td>
<td>55</td>
</tr>
<tr>
<td>6. Port f</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>7. Port g</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>8. Port h</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>9. Port i</td>
<td>25</td>
<td>35</td>
</tr>
<tr>
<td>10. Port j</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

Dataset2: Resources

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Category</th>
<th>MaxCapacity</th>
<th>Earliest Date Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>big</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>t3</td>
<td>big</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>t4</td>
<td>medium</td>
<td>60</td>
<td>0</td>
</tr>
</tbody>
</table>

Dataset2: Infeasible Resource-Task Allocation Constraints

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Port</th>
<th>Port</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>c</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>e</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Dataset2: Connectivity matrix

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>h</th>
<th>g</th>
<th>f</th>
<th>e</th>
<th>d</th>
<th>c</th>
<th>b</th>
<th>a</th>
<th>s1</th>
<th>s2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>s1</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>s2</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>e</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>f</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>g</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

Dataset2: A Generated Schedule with cost 40325 is:

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Size</th>
<th>Route Distance</th>
<th>Start Day</th>
<th>Ports On Route</th>
<th>Arrival (Day)</th>
<th>Min Required</th>
<th>Quantity Delivered</th>
<th>Day Avail for Next Voyage</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>33</td>
<td>0</td>
<td>a</td>
<td>8</td>
<td>45</td>
<td>46</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td>16</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b</td>
<td>24</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>big</td>
<td>32</td>
<td>0</td>
<td>d</td>
<td>12</td>
<td>45</td>
<td>45</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g</td>
<td>20</td>
<td>25</td>
<td>15(^1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
<td>24</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>big</td>
<td>28</td>
<td>0</td>
<td>h</td>
<td>5</td>
<td>30</td>
<td>18(^1)</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>j</td>
<td>13</td>
<td>30</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i</td>
<td>21</td>
<td>25</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>med</td>
<td>28</td>
<td>0</td>
<td>g</td>
<td>10</td>
<td>25</td>
<td>10</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>e</td>
<td>16</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>h</td>
<td>23</td>
<td>30</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

MultiDelivery to Port
Dataset2 Results:

The number of tasks in Dataset2 is 10, there are 3 tankers t1, t2 and t3 which have a big capacity. Tanker t4 has a medium capacity. Tanker t4 is the maximum constrained resource, and it cannot travel to ports c, a and b. A schedule generated with a cost of 43025 is provided as a sample schedule. The higher demand tasks scheduled first heuristic along with most constrained resource allocated first performed better than dispatching tasks using the same heuristic and allocating resources with larger capacity first, it could provide schedules in the cost range of below 36000 while the higher demand tasks scheduled first, along with resources with larger capacity allocated first could only provide a schedule with a minimum cost of about 37000. Task scheduling heuristic dispatching tasks which are farther first did better when compared to dispatching tasks with higher demand, it could provide the optimal solution in about 170-180 iterations compared to 230 iterations taken by dispatch higher demand tasks first heuristic.
Task Ordering Heuristics: Higher Demand Tasks Scheduled First
Resource Ordering Heuristics: More Constrained Resources Allocated first

Plot of Schedule Cost vs Iterations

Figure 5-9: Plots a,b for Tanker Scheduler Dataset 1
Figure 5-10: Plots c,d for Tanker Scheduler Dataset 1
Figure 5-11: Plots a,b for Tanker Scheduler Dataset 2
Figure 5-12: Plots c,d for Tanker Scheduler Dataset 2
Dataset3: Tasks

<table>
<thead>
<tr>
<th>Port</th>
<th>Q_{low} (10^6 brls)</th>
<th>Q_{high} (10^6 brls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Port a</td>
<td>45</td>
</tr>
<tr>
<td>2.</td>
<td>Port b</td>
<td>22</td>
</tr>
<tr>
<td>3.</td>
<td>Port c</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>Port d</td>
<td>45</td>
</tr>
<tr>
<td>5.</td>
<td>Port e</td>
<td>35</td>
</tr>
<tr>
<td>6.</td>
<td>Port f</td>
<td>20</td>
</tr>
<tr>
<td>7.</td>
<td>Port g</td>
<td>25</td>
</tr>
<tr>
<td>8.</td>
<td>Port h</td>
<td>30</td>
</tr>
<tr>
<td>9.</td>
<td>Port i</td>
<td>25</td>
</tr>
<tr>
<td>10.</td>
<td>Port j</td>
<td>30</td>
</tr>
<tr>
<td>11.</td>
<td>Port k</td>
<td>30</td>
</tr>
<tr>
<td>12.</td>
<td>Port l</td>
<td>25</td>
</tr>
<tr>
<td>13.</td>
<td>Port m</td>
<td>28</td>
</tr>
<tr>
<td>14.</td>
<td>Port n</td>
<td>31</td>
</tr>
<tr>
<td>15.</td>
<td>Port o</td>
<td>41</td>
</tr>
</tbody>
</table>

Dataset3: Infeasible Resource-Task Allocation Constraints

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Port</th>
<th>Port</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>c</td>
<td>e</td>
<td>g</td>
</tr>
<tr>
<td>t3</td>
<td>e</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>t5</td>
<td>f</td>
<td>g</td>
<td></td>
</tr>
</tbody>
</table>

Dataset3: Resources

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Category</th>
<th>MaxCapacity</th>
<th>Earliest Date Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>t2</td>
<td>big</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>t3</td>
<td>big</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>t4</td>
<td>big</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>t5</td>
<td>big</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>t6</td>
<td>medium</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>t7</td>
<td>medium</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>t8</td>
<td>small</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>
Dataset3: Connectivity Matrix

<table>
<thead>
<tr>
<th>s1</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>0</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>18</td>
<td>10</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>0</td>
<td>u</td>
<td>8</td>
<td>7</td>
<td>u</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>g</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>0</td>
<td>u</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>u</td>
<td>11</td>
<td>7</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>u</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>11</td>
<td>10</td>
<td>12</td>
<td>9</td>
<td>9</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>u</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>14</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>14</td>
<td>u</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>14</td>
<td>4</td>
<td>11</td>
<td>8</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>f</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>u</td>
<td>12</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>g</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>u</td>
<td>12</td>
<td>13</td>
<td>10</td>
<td>12</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>h</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>11</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>u</td>
<td>0</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>14</td>
<td>u</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>8</td>
<td>14</td>
<td>7</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>j</td>
<td>10</td>
<td>14</td>
<td>u</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>12</td>
<td>13</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>u</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>k</td>
<td>7</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>14</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>l</td>
<td>8</td>
<td>12</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>u</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>m</td>
<td>5</td>
<td>7</td>
<td>u</td>
<td>12</td>
<td>7</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>n</td>
<td>12</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>13</td>
<td>8</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>u</td>
</tr>
<tr>
<td>o</td>
<td>13</td>
<td>11</td>
<td>5</td>
<td>8</td>
<td>u</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>

Dataset3: A Generated Schedule with cost 63555 is:

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Size</th>
<th>Route Distance</th>
<th>Start Day</th>
<th>Ports On Route</th>
<th>Arrival (Day)</th>
<th>Min Required</th>
<th>Quantity Delivered</th>
<th>Day Avail for Next Voyage</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>33</td>
<td>4</td>
<td>a</td>
<td>12</td>
<td>45</td>
<td>46</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td>20</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b</td>
<td>28</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>big</td>
<td>34</td>
<td>3</td>
<td>h</td>
<td>8</td>
<td>30</td>
<td>30</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td>16</td>
<td>45</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i</td>
<td>30</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>big</td>
<td>39</td>
<td>7</td>
<td>j</td>
<td>17</td>
<td>30</td>
<td>30</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k</td>
<td>25</td>
<td>30</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>l</td>
<td>32</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m</td>
<td>41</td>
<td>28</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>big</td>
<td>32</td>
<td>0</td>
<td>e</td>
<td>14</td>
<td>35</td>
<td>35</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g</td>
<td>20</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
<td>24</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>big</td>
<td>32</td>
<td>0</td>
<td>m</td>
<td>5</td>
<td>28</td>
<td>18</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td>12</td>
<td>31</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>o</td>
<td>19</td>
<td>41</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>
Dataset3: Results

Dataset 3 requires 15 tasks to be scheduled, and there are 8 resources $t1$ through $t8$ which can be allocated to these 15 tasks. Tanker $t2$ and $t4$ are the most constrained tankers which cannot enter ports $\{c, e, g\}$ and $\{c, a, b\}$ respectively. The connectivity matrix for dataset3 contains 256 entries, with at least 10 infeasible route segments. A generated schedule with cost 63555 has been provided as a sample along with dataset3 information. This schedule used only 5 tankers $t1$ through $t4$ for scheduling purposes. The task dispatch heuristic - schedule Higher demand tasks scheduled along with the resource allocation heuristic most constrained resource allocated first, proved to be the best heuristic combination in this example, it provided solutions with costs of about 56,000 cost units in the range of 350-280 iterations. The minimum solution provided by the task heuristic Higher Demand Tasks Scheduled First and Resources with Larger capacity allocated first provided with solutions of higher costs within the same range of iterations. The heuristic combination of farther tasks scheduled first with more constrained resources allocated first proved to be better than the combination of farther tasks scheduled first and resources with larger capacity allocated first. As an illustration the first heuristics pair produced solutions with a cost below 60,000 in the first 50 iterations, while the other second heuristic pair could produce a solution with a lower value of 64,000 units within the same number of iterations.
Figure 5-13: Plots a,b for Tanker Scheduler Dataset 3
Figure 5-14: Plots c,d for Tanker Scheduler Dataset 3
Dataset 4: Tasks

<table>
<thead>
<tr>
<th>Port</th>
<th>$Q_{low}^p$ ($10^6$ brls)</th>
<th>$Q_{high}^p$ ($10^6$ brls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Port a</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>2. Port b</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td>3. Port c</td>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>4. Port d</td>
<td>45</td>
<td>50</td>
</tr>
<tr>
<td>5. Port e</td>
<td>35</td>
<td>55</td>
</tr>
<tr>
<td>6. Port f</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>7. Port g</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>8. Port h</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>9. Port i</td>
<td>25</td>
<td>35</td>
</tr>
</tbody>
</table>

Dataset 4: Resources

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Category</th>
<th>MaxCapacity</th>
<th>Earliest Date Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>big</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>t3</td>
<td>big</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>t4</td>
<td>big</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>t5</td>
<td>big</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>t6</td>
<td>small</td>
<td>40</td>
<td>0</td>
</tr>
</tbody>
</table>

Dataset 4: Infeasible Resource-Task Allocation Constraints

NONE.

Dataset 4: Grouping by using Meta Knowledge

Group 1,2,3 categorize ports based on proximity.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>a</td>
<td>d</td>
<td>g</td>
</tr>
<tr>
<td>Group 2</td>
<td>b</td>
<td>e</td>
<td>h</td>
</tr>
<tr>
<td>Group 3</td>
<td>c</td>
<td>f</td>
<td>i</td>
</tr>
</tbody>
</table>
```
Dataset4: Connectivity Matrix

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
<td>0</td>
<td>17</td>
<td>18</td>
<td>4</td>
<td>17</td>
<td>15</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>17</td>
<td>0</td>
<td>14</td>
<td>16</td>
<td>4</td>
<td>14</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>12</td>
<td>18</td>
<td>14</td>
<td>0</td>
<td>12</td>
<td>14</td>
<td>5</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td>4</td>
<td>16</td>
<td>12</td>
<td>0</td>
<td>18</td>
<td>13</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>e</td>
<td>12</td>
<td>17</td>
<td>4</td>
<td>14</td>
<td>18</td>
<td>0</td>
<td>13</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>13</td>
<td>15</td>
<td>14</td>
<td>5</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>g</td>
<td>14</td>
<td>6</td>
<td>17</td>
<td>14</td>
<td>16</td>
<td>16</td>
<td>5</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>h</td>
<td>13</td>
<td>15</td>
<td>6</td>
<td>12</td>
<td>16</td>
<td>5</td>
<td>14</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>12</td>
<td>16</td>
<td>16</td>
<td>6</td>
<td>13</td>
<td>15</td>
<td>4</td>
<td>9</td>
<td>14</td>
</tr>
</tbody>
</table>

Dataset4: A Generated Schedule with cost 42705 is:

<table>
<thead>
<tr>
<th>Tanker</th>
<th>Size</th>
<th>Route Distance</th>
<th>Start Day</th>
<th>Ports On Route</th>
<th>Arrival (Day)</th>
<th>Min Required</th>
<th>Quantity Delivered</th>
<th>Day Avail for Next Voyage</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>big</td>
<td>33</td>
<td>0</td>
<td>a</td>
<td>10</td>
<td>40</td>
<td>40</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td>14</td>
<td>45</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g</td>
<td>19</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>big</td>
<td>49</td>
<td>0</td>
<td>b</td>
<td>12</td>
<td>22</td>
<td>23</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>e</td>
<td>16</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>h</td>
<td>21</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td>37</td>
<td>45</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>big</td>
<td>33</td>
<td>0</td>
<td>c</td>
<td>12</td>
<td>30</td>
<td>39</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
<td>17</td>
<td>20</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i</td>
<td>21</td>
<td>25</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>
Dataset 4: Results

Dataset 4 involves delivery of oil to 9 ports and there are 6 resource \( t1 \) through \( t6 \) which are available for delivering oil. The ports are grouped together into three groups based on their physical proximity. We try a new task dispatch heuristic which dispatches tasks based on their physical proximity rather than the actual distance of the task from the source or the amount of oil supply which is required by the ports. This task heuristic has been described in a greater detail in Subsection 5.6.1 and examples are provided in Figure 5-1. A sample schedule generated with a cost of 42705 cost units has been provided along with the dataset information. The schedule costs obtained by scheduling using the task heuristic clustered tasks with small demand tasks scheduled first, and task heuristic clustered tasks with higher demand tasks scheduled first were substantially lower. The savings in cost comes primarily from minimization of routing cost, as the same resource is used for delivery of oil to ports which are in close proximity. Most of the optimal solutions for this problem were obtained between schedules generated in the range of 1-50 iterations and the lowest cost solutions solutions were in the range of 38,000 to 40,000 cost units. In contrast dispatching tasks with higher demands first, or tasks which were located further produced solutions which were substantially higher in cost, the least cost being 44,000 which was produced almost after 150 iterations. The results obtained from this dataset go on to further show how dispatch heuristics are very important for obtaining low cost schedules.
Figure 5-15: Plots a,b for Tanker Scheduler Dataset 4
5.9 Conclusions

In this chapter developing schedulers for a class of scheduling problems called transportation schedulers have been discussed in detail. Transportation schedulers involve delivering material to a number of destinations subject to constraints. Instances of transportation schedulers are highly combinatorial problems, because a large number of routes may exist between the source and destination, and material may be transported from source to destination by many resource route combinations. An Operations Research formulation of the problem yields a non-linear mixed binary program, which is very difficult to solve. In contrast, we have advocated an Artificial Intelligence approach which finds good solutions via generate and test. The method described can be used for both land and water based transportation scheduling, but we have concentrated in describing one instance, called the tanker scheduling problem, where oil tankers must be scheduled on various routes to meet crude oil requirements of different refinery locations.

There is as usual no single best heuristic which can be used to schedule transporting vessels. The vital parameters which should be taken into account while developing any scheduler are

*The Destination Demands:* It is better in general to satisfy ports with higher demands first. This philosophy has been used in industry and simulation results show that the port demands should always be taken into consideration while developing new heuristics.

*Destination Distances:* The physical location of different ports with respect to the supply stations are very important. The tasks which are located at a farther distance are harder to schedule, and following the philosophy of trying to schedule more difficult tasks first, destination distances should always be considered before devising new scheduling strategies.

Routing Constraints Destinations with fewer routes always present delivery problems when scheduled towards the end, it is always better to schedule them earlier in the schedule.
Resource Capacities Resources with larger capacities should be generally allocated first to the tasks. This helps to get more destinations serviced by the same resource and also increases resource utilization.

Resource Constraint Several resource may be constrained by the number of destinations they can visit as well as routes they can take. Resources which are constrained can prove to be bottleneck resources and should be considered early in the scheduling process.
Figure 5-16: Plots c,d for Tanker Scheduler Dataset 4
Chapter 6

Timetabling Schedulers

"Time itself flows on with constant motion, just like a river: for no more than a river can the fleeting hour stand still. As wave is driven on by wave, and, itself pursued, pursues the one before, so the moments of time at once flee and follow, and are ever new."

-Ovid

6.1 Introduction

The timetabling problem has been of interest since the mid 1960's [Bro64], [Col64], [MW66], [Bus67], [SO74]. The problem is of interest even now [FG89], [DMV89], [dW85] largely due to its varied nature, complexity and last but not the least its large size. The timetabling problem can be defined as scheduling of a certain number of sessions which must be attended by a specific group of enrollees, over a definite period of time, requiring certain resources. Some common examples of timetabling problems are classroom scheduling, examination scheduling and conference scheduling, art film festival scheduling and sports event scheduling. In classroom sessions are classes which have to be attended by students who are the enrollees and the resources are teachers, rooms and teaching aids for the classroom to be held, examination scheduling is a minor variation, here the exams are to be scheduled only once. Art film festival scheduling and sports event scheduling have films and various

185
*sport events* as sessions which must be scheduled based on participant preference. The aim again is to maximize aggregate spectator satisfaction which in the context of timetabling schedulers amounts to minimizing scheduling conflicts.

Most timetabling problems assume that a scheduling network is established ahead of time and the setup consists of a limited number of locations, say 9 meeting rooms or 300 classrooms, each of which is available for a limited number of time periods, say 5 meeting sessions, 40 examination hours, or 7 class periods. Secondly they assume that the best solution to the scheduling problem is found by allocating activities with that framework in such a way as to minimize the number of conflicts between activities. Conflicts are determined by polling the individuals who are affected by the schedule concerning the activities in which they plan to take part, and conflict reduction has been the main focus of study in some papers [Gri70].

Some excellent survey papers appear in the area of examination timetabling. One notable paper is by Carter [Car86], and one by Werra [dW85] where they summarize many of the successful methods used for generating examination schedules. A wide variety of approaches have been proposed to solve the classroom scheduling problem. These range from an integer/linear programming approach to solve the classroom scheduling problem, a purely graph formulation of school scheduling, to one solving classroom scheduling problems as a constraint satisfaction problem. A brief summary of these approaches is presented in Section 6.2.

### 6.2 Survey of Timetabling Applications

Most educational institutions must schedule a set of examinations at the end of each session, or must schedule classes for a large number of students. Every year a large number of conferences take place in different areas and due to the large number of papers which need to be presented in a limited time, the sessions have to be scheduled in parallel in such a way that most people get to attend their favorite sessions. All these problems belong to the class of problems we commonly call as timetabling problems.

In examination and conference scheduling problems, the sessions have to be
scheduled only once and hence the problem is called 1-day or interval problem. The multiday scheduling or non-interval scheduling problem consists of classes meeting in the same room over several week days. In the literature only specific problems like examination or classroom scheduling have been addressed at a time, so this survey will also present different instances of timetabling problems separately. Conference and examination scheduling are almost identical, and the results which are applicable to examination scheduling are applicable to conference scheduling as well. The only difference is that in conference scheduling the enrollees may due to some reason not be able to attend some of their favorite sessions, but in examination scheduling a student cannot really miss his exams. Many of the papers surveyed include suggestions and special consideration applicable in a wide variety of situations. The practitioner will probably find that the best approach is to develop a composite algorithm utilizing a cross section of ideas appropriate to his/her specific needs.

One of the earliest published examples of examination timetabling is presented by Broder [Bro64]. His stated objective is to minimize the number of student conflicts. His algorithm is basically a “largest degree first” method; in case of ties, each course in the list is randomly assigned to one of the time periods that creates the fewest number of conflicts. The algorithm is run several times and the best run is selected. He observes that the same algorithm is easily adapted to minimize the number of times a student has (1) two examinations in the same day, (2) three final examinations in succession, (3) three final examination in two days. Broder's method is relatively simple and produced good results if there were not too many special constraints in the system.

In 1964 Cole published an algorithm [Col64] that had been implemented successfully at Leicester University in England. Cole introduced constraints that allow

- Certain sets of examinations must be consecutive.
- Precedence ordering between exams.
- Space constraints on room sizes.
- Certain examinations have to be scheduled only in the morning.
Probably the most important in Cole's paper is his use of a bit matrix to represent course conflicts. Element \((i,j)\) is “1” if courses \(i\) and \(j\) have any conflicts and “0” otherwise. He does not store the number of student conflicts. In 1968 Wood [Woo68] devised an examination scheduling algorithm that was implemented at the University of Manchester for more than 1000 courses, 6000 students and 30 examination periods. Wood's problem had to schedule examinations into a set of designated rooms with limited capacities. For this reason he sorted the courses according to the size of room required. Within each size group he used the “largest degree first” rule. For each course in the list he searched for a feasible period with:

- No adjacent conflict, or, if none;
- no conflict on the same day ( 2 examinations per day) or , if none;
- the minimum number of students with another examination on the same day.

Some of the important features and results which Wood obtained were that “Courses should be scheduled starting with the most difficult course first”. The scheduler should define “most difficult” in relation to the feasibility of satisfying each constraint. In 1978, Barham and Westwood [BW78] presented an algorithm for timetabling courses under an elective system for the Manchester School of business. The problem involved a small group of 36 students registering for 22 optional courses to be taken in 40 sessions, with each course requiring between 3 and 6 sessions. Each student selected 5 or 6 courses. Barham's and Westwood’s approach is relevant to examination scheduling because they decided to construct a conflict free schedule based on each student’s first four selections. Students then would adjust their fifth and sixth electives, if necessary, after the fact. The proposed algorithm was simply a “largest degree first: fill from top” method. The results are interesting as an illustration of the use of a pure coloring algorithm on a difficult problem, the coloring problem defined in Graph Theory literature simply asks the minimum number of colors required to color nodes in a graph such that no adjacent nodes have the same colors.

Another notable paper in this is by Carter 1983 [Car83] who developed an algorithm for final examination scheduling at the University of Waterloo. In the case
of ties, a frequent occurrence in the recursive model, preference is given to large enrollment courses and then to certain special courses designated as “preferred” by the faculties. The major constraints were

- Several courses must be preassigned to fixed time periods.
- No student should be required to sit for three or more consecutive examinations;
- Certain examinations are designated as evening or Saturday only.

The second restriction was not implemented literally but a more restrictive rule disallowed scheduling any examinations that had conflicts in the two previous periods.

Classroom scheduling problem has been modeled as a large binary integer linear program by Arbinda Tripathy [Tri84], as a constraint satisfaction problem by Feldman et al. [FG89] where they propose constraint satisfiability algorithms for interactive student scheduling, and as a graph formulation by Salazar and Oakford [SO74]. In the current system we have used dispatch heuristics to schedule tasks in the timetabling domain rather than more mathematical solutions as appearing in [Tri84].

6.3 A Simple Timetabling Problem Formulation

The usual formulation of the problem is expressed in terms of the variable $X_{ij}$ a 0-1 valued variable which is equal to 1 when class $i$ is assigned to room $j$ and is 0 otherwise. Time is divided into periods (e.g. every half hour though all periods do not have to be of the same length). The number of periods are fixed to $k$ and the problem is called a $k$ period problem, $m$ is the total number of rooms available for assignment. The constraints in this problem are

$$\sum_{j=1}^{m} X_{ij} = 1 \text{ for } i = 1, 2, \ldots, n$$

$$\sum_{i \in P_k} X_{ij} \leq 1 \text{ for } j = 1, 2, \ldots, m; \ \forall k;$$
where \( P_k \) represents the set of all sessions that meet in period \( k \). The first constraint assures that every session is assigned to some room, while the second constraint prevents any room from having more than one session in the same period. We define three objectives for this problem:

Solve: Is there a feasible solution to the constraints above?

Satisfice: Given for each session \( i \), a set of rooms \( S_i \) that are satisfactory, is there a feasible solution to the constraint above that puts each session in a satisfactory room?

Optimize: Assume the existence of a linear cost model that adequately describes the problem and reflects reasonable preferences. The cost of assigning session \( i \) to room \( j \) is denoted by \( C_{ij} \). The cost could be reflected in terms of a dollar value associated with occupying a particular resource, or in terms of the conflicts the particular assignment would cause during the schedule. In the current implementation of timetable schedulers the cost \( C_{ij} \) has been taken to be equivalent to the number of conflicts an assignment of resource \( i \) to session \( j \) produces.

The objective is to minimize:

\[
\sum_{i=1}^{n} \sum_{j=1}^{m} C_{ij} X_{ij}
\]

The three objectives are listed in non-decreasing order of difficulty, regardless of other problem characteristics. This is because Feasibility can be modeled as a special case of Optimize. In particular if we assign a cost of 0 to all rooms \( j \) in the set \( S_i \), and a cost of 1 elsewhere, then Optimize gives the Satisfice solution if the final cost equals zero. Of the three objectives, the one closest to problems in practice is probably Satisfice. Manual schedulers tend to think in terms of "acceptable" room assignments.

Some of the implicit constraints are that a session is assigned to only one room, and there are two versions of the problem depending on how the word sessions is interpreted. 1-day version is called the interval scheduling problem. We call the first version the 1-day or interval problem. Ordinarily, sessions meet once in a day, from
the beginning of one period to the beginning of some another period. The multiday scheduling or non-interval problem consists of sessions meeting in the same room over several week days.

The simple timetabling problem is equivalent to the vertex coloring problem in graph theory. The latter problem has been studied extensively, and a wide variety of heuristics is available in the literature. This equivalence is particularly relevant for two reasons. First most of the timetabling heuristics are based on vertex coloring heuristics; and second, unlike the situation with timetabling several authors have presented computational results comparing the performance of the various vertex coloring algorithms. These results provide a partial vehicle for connecting and comparing timetabling applications.

6.4 Vertex Coloring Problem

The problem of finding a conflict-free timetable is structurally similar to the vertex coloring problem studied extensively in graph theory literature. For a given examination timetabling problem, a graph is constructed as follows:

- Each course is represented by a vertex.
- An edge connects two vertices if the corresponding courses have at least one student in common and hence cannot be scheduled in the same time period.

The graph coloring problem is usually posed as a question. Can the vertices of a graph be colored using a set of $p$ colors so that no two vertices connected by the same edge are assigned the same color? The analogy with the examination timetabling is completed by associating the $p$ exam periods to $p$ "colors." This unique minimum denoted by $\chi$, is called the chromatic number of the graph, and the problem of computing $\chi$ is known to be NP-Complete [Kar72].

The implications for examination timetabling depend on the structure of the particular problem. If the number of periods $p$ is much larger than $\chi$, then the problem of assigning $p$ conflict-free periods becomes relatively easy.

Grimmett and McDiarmid [GM77] have shown that for large random graphs, the simplest graph coloring heuristic will "almost always" use at most $2\chi$ colors, it is
likely that for a graph coloring in which \( p > 2\chi \), most heuristics will be able to find conflict-free schedules. Carter [Car86] has presented some evidence to suggest that timetabling problems are, in some sense, easier to solve than more general random graphs.

Practical examination timetabling problems differ from graph coloring when the following types of secondary constraints are added on the use of periods:

- A limit on the number of students and/or examinations in any one period;
- Room capacity constraints;
- Consecutive examination constraints (i.e. certain exams must occur in adjacent time periods).
- Non-consecutive conflict constraints (e.g. no examinations in succession for any students);
- Pre-Assignments (i.e. Certain examinations are preassigned to specific periods);
- Exclusions and time preferences (i.e. certain examinations are excluded from particular periods).
- Each student's examination should be evenly spaced over the examination period.

If \( p \) is is much greater than \( 2\chi \) then there is likely to be considerable flexibility in accommodating secondary constraints. If \( p \) is close to \( \chi \), then finding a conflict-free schedule becomes the primary objective and secondary constraints will typically be violated.

6.5 Timetabling Cluster Design

After a brief introduction to some of the approaches which people have taken to solve the timetabling problem, the data organization and current implementation of the timetabling scheduler is described under the generic scheduling framework. The
basic information is provided in forms on which candidates register for the examinations, enroll for certain classes or sessions. The format of schedule requirements is thus in the form of a person with a list of sessions he/she would like to enroll in preferences.

enroll([name::al,sessions::[101,102,103,108]]).
enroll([name::bert,sessions::[105,101,107,106]]).
enroll([name::chuck,sessions::[106,108,101,102,109]]).
enroll([name::dick,sessions::[101,111,104,110,108]]).
enroll([name::eva,sessions::[104,107,109]])..
enroll([name::faye,sessions::[102,107,108]])..

The following facts indicate the session requirements of different sessions, the main fields involved are SessionId(id), Duration(dur), Earliest Start Time(est), and the Latest End Time(LET). The earliest start time indicates a constraint where the particular session is required to be scheduled at a time no lesser than that indicated. The duration of all the sessions may not be similar and may be different as indicated in the example sample requirements. Many of the references considered always considered all the sessions to be of equal length, which is very restrictive. In the present example sessions with different durations are considered.

session_req([id::101,dur::3,est::5,let::15]).
session_req([id::102,dur::2,est::null,let::15]).
session_req([id::103,dur::2,est::null,let::15]).
session_req([id::104,dur::2,est::8,let::15]).
session_req([id::105,dur::1,est::4,let::15]).
session_req([id::106,dur::2,est::null,let::15]).
session_req([id::107,dur::3,est::5,let::15]).
session_req([id::108,dur::2,est::null,let::15]).
session_req([id::109,dur::2,est::5,let::15]).
session_req([id::110,dur::2,est::null,let::15]).
session_req([id::111,dur::2,est::9,let::15]).
Information about different sessions offered is obtained from the enrollment list provided. A list of all the sessions is provided below:


### 6.5.1 Tasks

The tasks in the system are also constructed from the enrollment data provided as input to the system. The task fields include CourseId (id), Duration(dur), Earliest Start Time (est), Latest End Time (let), Number of People Enrolled(n.enr), Number of Conflicts(n.conf), Start Time(st.time), end.time(end.time).

<table>
<thead>
<tr>
<th>CourseId</th>
<th>dur</th>
<th>est</th>
<th>let</th>
<th>n.enrolled</th>
<th>n.conflicts</th>
<th>st.time</th>
<th>end.time</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>3</td>
<td>5</td>
<td>15</td>
<td>4</td>
<td>14</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>null</td>
<td>15</td>
<td>3</td>
<td>9</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>103</td>
<td>2</td>
<td>null</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>104</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>2</td>
<td>6</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>null</td>
<td>15</td>
<td>2</td>
<td>7</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>107</td>
<td>3</td>
<td>5</td>
<td>15</td>
<td>3</td>
<td>7</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>108</td>
<td>2</td>
<td>null</td>
<td>15</td>
<td>4</td>
<td>13</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>109</td>
<td>2</td>
<td>5</td>
<td>15</td>
<td>2</td>
<td>6</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>null</td>
<td>15</td>
<td>1</td>
<td>4</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>111</td>
<td>2</td>
<td>9</td>
<td>15</td>
<td>1</td>
<td>4</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

The Prolog representation of different tasks is provided below.

```prolog
task([id::101,dur::3,est::5,let::15,n.enr::4,n.conf::14,s.tim::null,e.tim::null]).

task([id::102,dur::2,est::null,let::15,n.enr::3,n.conf::9,s.tim::null,e.tim::null]).

task([id::103,dur::2,est::null,let::15,n.enr::1,n.conf::3,s.tim::null,e.tim::null]).
```

### 6.5.2 Resources

The resources in the system are the rooms having fixed capacity and available for different periods of time. The main fields in each resource are ResourceId(id),
**Capacity(cap), Available Time (av.time)**. As the basic scheduling proceeds the availability time is updated, because the current room cannot hold more than one session at the same time.

```
resource([id::rm_1,cap::5,av.tim::[(1,20)]]).
resource([id::rm_2,cap::4,av.tim::[(1,20)]]).
resource([id::rm_3,cap::8,av.tim::[(1,20)]]).
resource([id::rm_4,cap::3,av.tim::[(1,20)]]).
resource([id::rm_5,cap::4,av.tim::[(1,20)]]).
resource([id::rm_6,cap::2,av.tim::[(1,20)]]).
```

### 6.5.3 Enrollment

<table>
<thead>
<tr>
<th>CourseId</th>
<th>Num Enrolled</th>
<th>Enrolee</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>4</td>
<td>al,bert,chuck,dick</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>al,chuck,faye</td>
</tr>
<tr>
<td>103</td>
<td>1</td>
<td>al</td>
</tr>
<tr>
<td>104</td>
<td>2</td>
<td>dick,eva</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>bert</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>bert,chuck</td>
</tr>
<tr>
<td>107</td>
<td>3</td>
<td>bert,eva,faye</td>
</tr>
<tr>
<td>108</td>
<td>4</td>
<td>al,chuck,dick,faye</td>
</tr>
<tr>
<td>109</td>
<td>2</td>
<td>chuck,eva</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>dick</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>dick</td>
</tr>
</tbody>
</table>

The enrollment in different courses is an inverse transformation from the data provided, where preferences of different people enrolled are provided.

```
enroll(101,4,[al,bert,chuck,dick]).
enroll(102,3,[al,chuck,faye]).
enroll(103,1,[al]).
```
6.5.4 Constraints

The constraints in the system can be divided into resource and temporal constraints, in a manner very similar to the carshop scheduling. The temporal constraints in the system can be either explicit or implicit, one of the implicit temporal constraints in the system is that two sessions/courses/meetings cannot be scheduled at the same time if they have any common enrollees. These two sessions can be thought of as adjacent vertices in a graph, which cannot be colored using the same color i.e. scheduled in the same time period.

% pre_assign(Course,Start_time,Room).
% time or room can be null, indicating no fixed preference
   pre_assign(107,5,null).
   pre_assign(108,null,rm_4).
   pre_assign(110,6,rm_6).
% forbidden_assignment(Course,Room).
   forbidden_assignment(104,rm_3).
   forbidden_assignment(108,rm_6).

6.6 Conflict List

The conflict matrix \( cft \) is built where

\[ cft(i, j) = \text{number of candidates taking both subjects } i \text{ and } j. \]

\[ cft(i, i) = 0 \forall i. \]

A more efficient representation of the conflict matrix is in the form of a conflict list. The conflict list is a dynamic structure where dynamically the number of conflicts between different sessions are kept. A large amount of space is conserved by such an arrangement although accessing the conflicts takes more time. Both of these representations appear in the next page.

A corresponding Prolog representation of the individual conflicts between different sessions follows after the conflict matrix and list representations.
Conflict Matrix:

<table>
<thead>
<tr>
<th></th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
<th>111</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>103</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>107</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>108</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Conflict List:

<table>
<thead>
<tr>
<th>CourseId</th>
<th>N_Conflicts</th>
<th>Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>14</td>
<td>(102,2),(103,1),(104,1),(105,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(106,2),(107,1),(108,3),(109,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(110,1),(111,1)</td>
</tr>
<tr>
<td>102</td>
<td>9</td>
<td>(101,2),(103,1),(106,1),(107,1),(108,3),(109,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(101,1),(102,1),(108,1)</td>
</tr>
<tr>
<td>103</td>
<td>3</td>
<td>(101,1)</td>
</tr>
<tr>
<td>104</td>
<td>6</td>
<td>(101,1),(107,1),(108,1),(109,1),(110,1),(111,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(101,1),(106,1),(107,1)</td>
</tr>
<tr>
<td>105</td>
<td>3</td>
<td>(101,2),(102,1),(105,1),(107,1),(108,1),(109,1)</td>
</tr>
<tr>
<td>106</td>
<td>7</td>
<td>(101,1),(102,1),(104,1),(105,1),(106,1),(108,1)</td>
</tr>
<tr>
<td>107</td>
<td>7</td>
<td>(101,3),(102,3),(103,1),(104,1),(106,1),(107,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(109,1),(110,1),(111,1)</td>
</tr>
<tr>
<td>108</td>
<td>13</td>
<td>(101,1),(102,1),(104,1),(106,1),(107,1),(109,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(110,1),(111,1)</td>
</tr>
<tr>
<td>109</td>
<td>6</td>
<td>(101,1),(102,1),(104,1),(106,1),(107,1),(108,1)</td>
</tr>
<tr>
<td>110</td>
<td>4</td>
<td>(101,1),(104,1),(108,1),(111,1)</td>
</tr>
<tr>
<td>111</td>
<td>4</td>
<td>(101,1),(104,1),(108,1),(110,1)</td>
</tr>
</tbody>
</table>
In the first fact session 101 has 14 conflicts out of which session 102 contributes 2, session 103 1 and so on. This essentially means that 2 people enroll in session 101 and 102, 1 person enrolls in 101 and 104.

\[
\text{conf}(101,14,[(102,2),(103,1),(104,1),(105,1),(106,2),(107,1),
(108,3),\ldots (111,1)])
\]

\[
\text{conf}(102,9,[(101,2),(103,1),(106,1),(107,1),(108,3),(109,1)])
\]

\[
\text{conf}(103,3,[(101,1),(102,1),(108,1)])
\]

### 6.7 Timetabling Schedule Generation

The algorithm presented in Figure 6-5 is a specialized instance of the more general algorithm presented in Figure 3-11. The tasks in the timetabling algorithm are sessions which need to be scheduled, and the resources are the venues where the task needs to be scheduled. The algorithm in Figure 6-5 is general enough to handle the case of both interval and non-interval schedulers. Using the generate and test methodology which has been proposed in this thesis, scheduling proceeds in the following stages:

#### 6.7.1 Initialization

This is a part of any schedule generation and it sets the values of variables involved in scheduling to proper initial values. Specifically initialization involves collecting all the sessions which need to be scheduled, calculating the enrollment of each session, generating all the conflicts which arise between all the sessions, generating all the tasks based on the session, enrollment and conflict information. Next certain variables, namely number of conflicts \( n_{\text{conflicts}} \), current\.tasks current\.resources, allocation, iterations, perm\.count are reset. When multiple iterations of the same schedule are performed the predicate re\.initialize is invoked, and this resets the number of conflicts and current allocation allocation in the schedule. Assert and retract are used while collecting sessions, conflicts, enrollment, tasks because this information is used in many predicates and is easier to read it from memory rather than passing this information along.
6.7.2 Task Selection

The task to be scheduled is selected in step 1 of the algorithm. Each task has a session identifier and is selected based on the heuristics specified. Task ordering may be static or dynamic depending on the heuristic type used. If task selection is done based on a static heuristic then tasks are ordered only once in the beginning of schedule generation but a dynamic selection heuristic needs task ordering to be re-evaluated at the end of each schedule cycle. The `select/3` predicate is used for task selection, the list of current tasks is asserted into the Prolog database at the beginning of each schedule cycle. Other task heuristics like `select_shortest_task_first` require that all tasks be collected and sorted once depending on their duration.

6.7.3 Resource Selection

Resource allocation is dependent on the task which was selected to be scheduled. The sample Prolog code for `choose_update_resource/6` is an example of a particular resource selection procedure which allocates the earliest avail-
choose_task(Task, Tasks, RestTasks) :-
    current_task_hs(least_scheduleable_first),
    order_least_interval_task_first(Tasks, OrderedTasks),
    select(Task, OrderedTasks, RestTasks),
    abolish(current_tasks/1),
    assert(current_tasks(RestTasks)),
    !.

% for all heuristics which have a static re-ordering of tasks.
choose_task(Task, Tasks, RestTasks) :-
    select(Task, Tasks, RestTasks),
    abolish(current_tasks/1),
    assert(current_tasks(RestTasks)),
    !.

% task ordering is done by a sort function which sorts a list
% of objects based on specified attributes.

order_tasks(most_conflicts_first, Tasks, OrderedTasks) :-
    sort_list_form([(standard, n_conflicts, dsc), [], Tasks, OrderedTasks]).

order_tasks(shortest_task_first, Tasks, OrderedTasks) :-
    sort_list_form([(standard, duration, asc), [], Tasks, OrderedTasks]).

order_tasks(longest_task_first, Tasks, OrderedTasks) :-
    sort_list_form([(standard, duration, dsc), [], Tasks, OrderedTasks]).

Figure 6-2: Task Selection during Timetabling Schedule Generation

able resource to a task. First task parameters are obtained by accessing task fields. Then the earliest schedulable interval is generated during which the particular task can be scheduled, this interval is a conflict free interval, and the predicate generate_earliest_interval can backtrack to generate all possible conflict free schedulable intervals during which the task can be scheduled. An appropriate room/venue is selected for the task where it is made certain that the venue/room has a capacity equal to or greater than that required by the task which has been scheduled, and there are no constraints which prevent the task/resource assignment.
check_update_resource.availability checks if the resource is available during the interval generated by generate.earliest.interval and updates the current resource availability if the selected task can indeed be scheduled during the specified time interval. In case the resource cannot be scheduled, choose.appropriate.room backtracks to generate the next appropriate room. If all possible venues have been exhausted and still no venue is found at which the session can be scheduled, the control flow backtracks to generate.earliest.interval when the next earliest interval for scheduling is generated. A sample piece of code for choosing and updation of resource using the earliest schedulable resource heuristic appears in Figure 6-3.

```
% resource selection is always dynamic and depends on the task selected to be scheduled
% The first predicate is an example of earliest schedulable resource first

choose.update_resource(e.sched.ros.task(Task,F),Rcs,ros(Rcs,F),S.Time,E.Time,UpdtRcs) :-
  access.Fld.val([sched.int,n.enr,dur]:Task,F,[Schedulable.Int,N.Enrolled,Dur]),
  generate.earliest.interval(Schedulable.Int,Dur,S.Time,E.Time),
  choose.appropriate.room(N.Enrolled,Rcs,ros(Rcs,F),RcsAvTime),
  check.update_resource.availability(S.Time,E.Time,RcsAvTime,Dur,UpdtRcsAvail),
  update.resource.list(Rcs,ros(Rcs,F),UpdtRcsAvail,UpdtRcs),
  !.

% The second predicate is invoked if there are no non-conflicting scheduling intervals. The % best thing to do is to look for a venue whose capacity is closest to the % task requirement and an interval where number of conflicts are minimal.

choose.update_resource(e.sched.ros.task(Task,F),Rcs,ros(Rcs,F),S.Time,E.Time,UpdtRcs) :-
  access.Fld.val([id,n.enr,dur]:Task,F,[TaskId,N.Enrolled,Dur]),
  choose.appropriate.room(N.Enrolled,Rcs,ros(Rcs,F),RcsAvTime),
  generate.least.conflicting.interval(TaskId,RcsAvTime,Dur,S.Time,E.Time,N.Confs),
  check.update_resource.availability(S.Time,E.Time,RcsAvTime,Dur,UpdtRcsAvail),
  update.resource.list(Rcs,ros(Rcs,F),UpdtRcsAvail,UpdtRcs),
  update.total.conflicts(N.Confs),
  !.
```

Figure 6-3: Choosing and Updating Resources during Schedule Generation

If the first predicate fails then there exists no resource which can schedule the task without introducing conflicts into the schedule. The second predicate is invoked in this case when there is no non-conflicting schedulable intervals for the task. The scheduling strategy now picks up the resource whose capacity is closest to
the task requirement and schedules the task when it is least conflicting with the earlier scheduled tasks. The predicate `generate_least_conflicting_interval` looks for the least conflicting interval \((S\_Time, E\_Time)\) between which the task can be scheduled in the current resource. If no interval is found then the predicate fails and `choose_appropriate_room` generates the next appropriate venue. If the current `choose_update_resource` fails, the task is unschedulable and the current schedule cycle fails.

### 6.7.4 Allocation

Allocation of resources is a step where bookkeeping is done. In the allocation phase the task list is updated and the schedule interval \((ST_i, ET_i)\) is removed from the resource availability. In the current implementation resource selection and allocation are done within the same Prolog predicate.

### 6.7.5 Constraint Propagation

Once a task is scheduled then it occupies a time duration \((ST_i, ET_i)\) which are the start and end times of the task. All enrollees who wish to enroll in this session as well as other sessions, should have the other sessions also called as conflicting sessions scheduled on a time interval not overlapping with \((ST_i, ET_i)\). The constraint propagation step precisely achieves this effect by removing the interval \((ST_i, ET_i)\) from the schedulable intervals of all conflicting tasks. In the next step of the iteration if one of these tasks is chosen for being scheduled then the resource allocator will search for a schedulable interval which excludes \((ST_i, ET_i)\) as this interval does not belong to the schedulable interval of the new task.

For non-interval schedulers the constraint propagation also needs to set the schedulable intervals of other tasks which have the same session identifier as the current task. An example of a non-interval scheduler is the classroom scheduler where more than one instance of a particular class has to be scheduled during a week and generally two classes of the same session should not be scheduled contiguous to one another. The constraint propagator prevents this by performing the operation shown in the next page.
% propagate constraints modifies schedulable intervals of conflicting % tasks.
propagate_constraints([],UpdatedTaskList) :-
    current_tasks(UpdatedTaskList).

% Propagating constraints of a Scheduled task, constraints of other scheduled % tasks RestSchedTasks are done in the next call to propagate_constraints.
propagate_constraints([ScheduledTask|RestSchedTasks],UpdatedTaskList) :-
    current_tasks(Tasks),
    update.schedulable_intervals([ScheduledTask,Tasks,[],UpdatedTasks),
    update.current_tasklist(UpdatedTasks),
    propagate_constraints(RestSchedTasks,UpdatedTaskList).

update.schedulable_intervals([Task,[],Acc,UpdatedTasks) :-
    reverse(Acc,UpdatedTasks).

update.schedulable_intervals(Task,[Task1|Rest1],Acc,UpdatedTasks) :-
    check_conflict(Task,Task1,Interval),
    !,
    update_task(Interval,Task1,UpdatedTask1),
    update.schedulable_intervals(Task,Rest1,[UpdatedTask1|Acc],UpdatedTasks).

% The next predicate is used for non-interval schedules only.
% It removes Interval Before and Interval After from any task with % the same session identifier.
update.schedulable_intervals(Task,[Task1|Rest1],Acc,UpdatedTasks) :-
    same_id(Task,Task1),
    !,
    generate_unschedulable_intervals(Task,IntervalBefore,IntervalAfter),
    update_task(IntervalBefore,Task1,T1),
    update_task(IntervalAfter,T1,UpdatedTask1),
    update.schedulable_intervals(Task,Rest1,[UpdatedTask1|Acc],UpdatedTasks).

update.schedulable_intervals(Task,[Task1|Rest1],Acc,UpdatedTasks) :-
    update.schedulable_intervals(Task,Rest1,[Task1|Acc],UpdatedTasks).

update.current_tasklist(UpdatedTasks) :-
    abolish(current_tasks/1),
    assert(current_tasks(UpdatedTasks)).

check_conflict(task(Task,F),task(Task1,F),(S.Time,E.Time)) :-
    access.Fid.val([id,sim,c.tim]:=Task,F,[Sched_Task.Id,S.Time,E.Time]),
    access.Fid.val([id]:=Task1,F,[Conf.Task.Id]),
    conflicts(ConflictMatrix),
    !,
    member(conf(Sched_Task.Id,_N.Sess,_N.Conf,ConflictingSessions),ConflictMatrix),
    member((Conf.Task.Id,_,ConflictingSessions).

Figure 6-4: Constraint Propagation during Timetabling Schedule Generation

Constraint Propagation:

\[
\forall \text{ Task } T_k \ni \text{ sessid}(T_k) = \text{ sessid}(T_i) \text{ and scheduled}(T_i) \]

Remove interval \((ST_i-T_i, MinimumGap, ST_i)\) and \((ET_i, ET_i + ST_i, MinimumGap)\)

from \(T_k, \text{ScheduleInterval}\)
Figure 6-5: “Generate and Test” Scheduling Algorithm Specialized for Timetabling Schedulers

If there is no such restriction in the case of a non-interval scheduler then $T_i.MinimumGap$ can be set to 0.
6.7.6 Calculations/Postprocessing
If a task cannot be scheduled in its schedulable hours, then an attempt is made to schedule it using a resource whose size is the nearest possible to the enrollment number of the session. During this process there are conflicts which are generated. The current task is scheduled during intervals where it conflicts least with other sessions. The scheduler also collects other statistics, which could be the total cost of assignment of resources to tasks. If all tasks have been scheduled then the scheduler stores the best solution generated so far. If the required number of iterations have been performed then the scheduling process stops and the current best solution is reported, else the next schedule is generated. In the current cluster the next schedule is generated by obtaining a permutation of tasks and restarting the scheduling process. Standard backtracking in Prolog is not used for generating a new schedule because it does not tend to produce sufficiently different schedules.

6.8 Timetabling Schedule Trace
In order to make the scheduling process clearer two example traces have been provided. Both of them are for interval schedulers, where the session has to be scheduled exactly once. The first example in Figure 6-9 require 5 sessions 101,102,103,104,105 to be scheduled. The durations of the sessions is also provided along with the people enrolled in the sessions. Three resources room rm_1, rm_2, rm_3 are available for allocation, and each have a capacity of 5, 4 and 8 respectively. The conflict matrix generated is presented in the same figure, sessions pairs (101,103), (101,102), (101,104) each have 3 conflicts. The conflicts for session pairs (101,103) arise because they have \{a, c, e\} as common enrollees, and scheduling 101 and 103 during the same time will cause a conflict of interest for each one of \{a, c, e\} who cannot attend two of the sessions they have enrolled in at the same time. The conflict matrix is the matrix representation of the conflict graph. Session 101 has 9 as the number of conflicts, this is the the highest number of conflicts, while session 105 has only 2 conflicts since only \{g, h\} are enrolled for both session 105 and 103.
6.8.1 Schedule Trace using Shortest Session First, Closest Fitting Resource First

Initially in Figure 6-7 the sessions 101-105 can be scheduled anywhere between intervals (1,6). The task ordering heuristic used in this case is shortest session first,
Figure 6-7: Schedule Generation using Shortest Task, Closest Fitting Rcs (1)

while the resource selection heuristic is choosing the closest fitting resource. The first task to be scheduled is 104 its duration is 1 time interval. The enrollment of 104 is 4 so resource Rm.2 with a capacity of exactly 4 is allocated to it. Once the
Figure 6-8: Schedule Generation using Shortest Task, Closest Fitting Rs (2)

allocation is done, the resource availability is updated, and constraint propagation removes interval (1,2) from tasks 101, 102, 103. Task 105 does not conflict with 104, hence its schedulable interval remains the same. This procedure is repeated
and tasks 105, 101, and 103 are scheduled in that order. At each stage the updated intervals have been shown, and after task 103 has been scheduled task 102 has no schedulable interval. This means that it cannot be scheduled during any time period without producing a conflict. Session 102 has an enrollment of 5, and Rm.1 is the resource with the closest capacity, so all total number of conflicts in all time intervals namely (1,4) and (2,5) when resource Rm.1 is available are calculated. The minimum number of conflicts is generated in the interval (2,5), so the task 102 is scheduled in the period (2,5). The shaded region in Figure 6-8 shows the periods and session with which 102 conflicts i.e. 101 and 103. The total number of conflicts are 4.

6.8.2 Schedule Trace using Most Conflicting Task First, Earliest Schedulable Resource First

In this example tasks are ordered based on the number of conflicts they have. Task 101 has the maximum number of conflicts which is 9 in number, while other tasks 103, 104, 102 and 105 have 8, 8, 7 and 2 conflicts respectively. The idea here is to try and schedule the difficult tasks first, after which the easier tasks are scheduled, the difficulty of a task is measured in terms of the number of conflicts it has. Once task 101 is scheduled in room Rm.3 (the only room which can hold it due to size and enrollment consideration), the schedulable intervals of 102, 103, 104 are immediately changed to period [(3,5)], as all these conflict with session 101. Sessions 103, 104 are scheduled next using the same procedure. Session 102 is the fourth task to be scheduled and it has no schedulable intervals present. Session 102 has an enrollment of 5 people and the only time it can be scheduled is between (3,6) using resource Rm.3. This creates 4 conflicts as shown in Figure 6-10. The last task to be scheduled is 105, this task has the least number of conflicts and also since its duration is only 1 time unit, it was scheduled in Rm.1 and produced no further increase in the number of conflicts.

6.9 Session Scheduling Simulation Results

In this section we present the results which were obtained by trying to apply an implementation of the algorithm to a series of scheduling problems and looking
Figure 6-9: Schedule Generation using Most Conflicting Task, Earliest Sched. Rcs (1)

at the results obtained. The Prolog program for generating session schedules is approximately 1000 lines. Eight problem sets were chosen and experiments were
Figure 6-10: Schedule Generation using Most Conflicting Task, Earliest Sched. Rcs (2)

done to see the performance of different heuristics. We present the sample input data of three problem sets and also explain the observations made during the scheduling
process. The three problems presented in detail are all interval scheduling problems where all sessions have to be scheduled once i.e it is a 1-day problem. All these examples are conference scheduling examples where a number of participants have to be enrolled for various sessions. There are two kinds of problems which have been considered here, in one problem the conflict graph is dense, i.e. there are many conflicting sessions while in the third problem the conflict matrix is partitioned into two components. These examples enable us to study the effect of the conflict matrix graph on the scheduling techniques used. The dataset is presented after the problem description.

6.9.1 Problem Descriptions
The first two problems presented in dataset 1 and 2 are instances of conference scheduling problems. Each have to schedule 50 enrollees who have enrolled for 20 sessions.

These sessions range from 101-120. The maximum number of sessions any single person is enrolled in is 6; enrollees with identifiers 1, 4, 16, 20 are examples of people who are enrolled in 6 sessions. The minimal number of sessions some enrollees are enrolled in is 2, enrollees 21, 26, 27 are examples of people enrolled in only two sessions.

The enrollment data is converted into tasks which have to be scheduled the values of whose fields are shown below the enrollment data. Each of the tasks has 10 fields, the fields are TaskId which is the task identifier, Dur which is the duration for which the task must be scheduled, EST the earliest start time of the task, LET the latest end time of the task,

N.ENC number of people enrolled for the particular session, Conflicting Sessions which is the number of sessions conflicting with the task, total conflicts which is the maximum number of people who would be affected if all the conflicting sessions involving the current task were scheduled at overlapping times, start, end are the start and end time of the sessions which has to be determined by the scheduler. Sched Interval is the basic interval in which the task can be scheduled without causing any conflicts. This interval keeps decreasing for all tasks as more and more
<table>
<thead>
<tr>
<th>Id</th>
<th>Sessions Enrolled</th>
<th>Id</th>
<th>Sessions Enrolled</th>
<th>Id</th>
<th>Sessions Enrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[120]</td>
<td></td>
<td></td>
<td>120</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[120]</td>
<td></td>
<td></td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[107,103,105,110]</td>
<td>21</td>
<td>[101,103]</td>
<td>38</td>
<td>[103,116]</td>
</tr>
<tr>
<td></td>
<td>[110]</td>
<td></td>
<td></td>
<td>110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[108]</td>
<td></td>
<td></td>
<td>108</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[119]</td>
<td></td>
<td></td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>[101,102]</td>
<td>50</td>
<td>[102,103,114,115]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Enrollment Data for Dataset 1,2

Tasks get scheduled.

**Dataset 1,2:** We now present a brief description of dataset 1 and 2, which differ only in their resource definitions. For dataset 1, 2 tasks are shown in Table 6.2. Tasks 101, 103, 105, 107 all have a very high number of conflicts which are in the range of 61 to 65. Some of the sessions like 110, 113, 116 have very less conflicts i.e. ranging from 14 to 17. These conflicts are a measure of the difficulty in scheduling the tasks, for example scheduling session 101 for a duration of 2 time units means that none of the 16 sessions which conflict with it can be scheduled at any overlapping time without causing any conflicts. Sessions 113, 114 have a low enrollment and consequently have a lesser number of conflicting sessions and conflicts. The number of people enrolled in a session also is very important as it determines the resource which has to be allocated to the task as well as the fact
<table>
<thead>
<tr>
<th>Id</th>
<th>Dur</th>
<th>EST</th>
<th>LET</th>
<th>N_ENR</th>
<th>Conflicting Sessions</th>
<th>Total Conf</th>
<th>Start</th>
<th>End</th>
<th>Schedulable Intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>18</td>
<td>16</td>
<td>61</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>13</td>
<td>27</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>103</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>18</td>
<td>61</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>104</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>17</td>
<td>14</td>
<td>56</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>17</td>
<td>16</td>
<td>61</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>6</td>
<td>11</td>
<td>26</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>107</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>17</td>
<td>14</td>
<td>65</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>108</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>14</td>
<td>30</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>109</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>14</td>
<td>14</td>
<td>49</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>110</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>13</td>
<td>41</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>13</td>
<td>26</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>113</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>114</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>115</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>17</td>
<td>48</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>116</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>5</td>
<td>10</td>
<td>17</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>117</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>15</td>
<td>31</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>118</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>5</td>
<td>12</td>
<td>20</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>119</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>10</td>
<td>15</td>
<td>38</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
<tr>
<td>120</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>12</td>
<td>25</td>
<td>null</td>
<td>null</td>
<td>(1,15)</td>
</tr>
</tbody>
</table>

Table 6.2: Tasks with All Fields - Dataset 1,2

that sessions with high enrollment also tend to have larger conflicts.

Resources in the case of dataset 1 and 2 are different, they are shown in Table 6.5. In dataset 1 all the resources are identical, each having a maximum capacity of 20 people. In dataset 2 resource capacities of different rooms are different. There is one venue having capacity 10, two venues having capacity 15 and another two having capacity 20. There are 5 tasks 101, 103, 104, 105, 107 each of which have an enrollment of more than 15 and 10 tasks 102, 106, 108, 110, 113, 114, 116, 117, 118, 120 each of which have an enrollment less than 10. Other 5 tasks have an enrollment between 10 and 15. All the resource availability are initially kept in the range $[(1,15)]$.

**Dataset 3:** In dataset 3 the objective is to schedule 60 enrollees to 21 sessions. One major difference between this dataset and the previous dataset is in the conflict graph. In dataset 3 the enrollees can be partitioned into 3 sets, the first
<table>
<thead>
<tr>
<th>Id</th>
<th>Conflicts</th>
<th>Conflicting Sessions</th>
<th>Total Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>[(102,5),(103,6),(104,6),(105,7),(106,3),(107,6)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(108,3),(109,4),(110,1),(111,5),(112,2),(115,4)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(119,5),(120,1),(116,1),(117,2)]</td>
<td>16</td>
<td>61</td>
</tr>
<tr>
<td>102</td>
<td>[(101,5),(103,1),(104,3),(105,4),(106,3),(107,2)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(109,1),(111,2),(112,1),(114,1),(115,2),(117,1)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(119,1)]</td>
<td>13</td>
<td>27</td>
</tr>
<tr>
<td>103</td>
<td>[(101,6),(102,1),(104,5),(105,6),(106,1),(107,8)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(108,3),(109,5),(110,2),(111,2),(112,1),(114,1)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(115,6),(116,4),(117,1),(118,3),(119,4),(120,2)]</td>
<td>18</td>
<td>61</td>
</tr>
<tr>
<td>104</td>
<td>[(101,5),(102,3),(103,5),(105,8),(106,4),(107,9)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(108,2),(109,6),(111,1),(113,1),(115,3),(117,3)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(118,2),(119,3)]</td>
<td>14</td>
<td>56</td>
</tr>
<tr>
<td>105</td>
<td>[(101,7),(102,4),(103,6),(104,8),(106,5)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(107,10),(108,4),(109,4),(110,1),(111,3),(112,1)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(115,1),(116,2),(117,3),(118,1),(120,1)]</td>
<td>16</td>
<td>61</td>
</tr>
<tr>
<td>106</td>
<td>[(101,3),(102,3),(103,1),(104,4),(105,5),(107,4)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(108,1),(109,2),(110,1),(111,1),(115,1)]</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>107</td>
<td>[(101,5),(102,2),(103,8),(104,9),(105,10),(106,4),(108,5)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(109,8),(110,2),(111,2),(115,5),(116,1),(118,1),(119,2)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>[(101,3),(103,3),(104,2),(105,4),(106,1),(107,5),(109,4)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(110,1),(111,1),(112,1),(113,1),(117,1),(119,2),(120,1)]</td>
<td>14</td>
<td>65</td>
</tr>
<tr>
<td>109</td>
<td>[(101,4),(102,1),(103,5),(104,6),(105,4),(106,2),(107,8)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(108,4),(110,3),(111,4),(112,1),(115,3),(116,1),(117,1)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(118,1),(119,1)]</td>
<td>16</td>
<td>49</td>
</tr>
<tr>
<td>110</td>
<td>[(101,1),(103,2),(105,1),(106,1),(107,2),(108,1),(109,3)]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[(111,1),(115,2)]</td>
<td>9</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 6.3: Conflict Matrix Generated for Dataset 1.2

The set consists of enrollees 101-107, the second one has enrollees 108-114 and the third one between 109-121, where the starting and ending enrollees identifiers are included in the set. There is an overlap of sessions between enrollees of the first set and the second set but the overlap is really minimal. Sessions 105, 106, 107 are some of the sessions which are taken by people enrolled in the second set. The enrollees in the third set do not interact with the people in the first two sets. The conflict matrix thus consists of two components. The first component consists of sessions between 101-114 while the second component consists of sessions between 114-121. The first component the graph can be further broken down into more components if sessions
105, 106 and 107 are removed from the graph. The conflicts in this examples are lesser for each session. Some sessions like 112, 119 have a lot of conflicts because they are core sessions which are taken by a majority of people.

The resource for dataset 3 consists of 5 rooms, the capacities of two of them is 10 while the capacity of the remaining three is 15. All resources are uniformly available during the time interval ([1,15]).

Two schedules generated based on dataset 3 have been presented, the second one is a partial schedule where some tasks remain to be scheduled. The current schedule
<table>
<thead>
<tr>
<th>Id</th>
<th>Conflicts</th>
<th>Conflicting Sessions</th>
<th>Total Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>(102,4),(103,4),(104,1),(105,2),(106,6),(107,5)</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>102</td>
<td>(101,4),(103,4),(104,4),(105,3),(106,6),(107,2)</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>103</td>
<td>(101,4),(102,4),(104,3),(105,3),(106,4),(107,5)</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>104</td>
<td>(101,4),(102,4),(103,3),(105,7),(106,4),(107,6)</td>
<td>11</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>(108,1),(110,2),(111,2),(112,1),(113,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>(101,2),(102,3),(103,3),(104,7),(106,4),(107,4)</td>
<td>11</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>(108,1),(110,2),(111,4),(112,2),(113,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>(101,6),(102,6),(103,4),(104,4),(105,4),(107,5)</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>(111,2),(112,1),(113,2),(114,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>(101,5),(102,2),(103,5),(104,6),(105,4),(106,5)</td>
<td>12</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>(108,2),(109,2),(110,2),(111,2),(112,2),(114,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>(104,1),(105,1),(107,2),(109,5),(110,4),(111,3)</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>(112,5),(113,1),(114,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>(107,2),(108,5),(110,4),(111,2),(112,8),(113,2),(114,5)</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>110</td>
<td>(104,2),(105,2),(107,2),(108,4),(109,4),(111,6)</td>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>(112,6),(113,1),(114,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>(104,2),(105,4),(106,2),(107,2),(108,3),(109,2)</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>(110,6),(112,4),(113,3),(114,2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>(104,1),(105,2),(106,1),(107,2),(108,5),(109,8),(110,6)</td>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>(111,4),(113,5),(114,8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>(104,1),(105,2),(106,2),(108,1),(109,2),(110,1),(111,3)</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>(112,5),(114,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>(106,1),(107,1),(108,4),(109,5),(110,3),(111,2)</td>
<td>8</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>(112,8),(113,3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>(116,4),(117,1),(118,3),(119,5),(120,6),(121,6)</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>116</td>
<td>(115,4),(117,1),(118,4),(119,2),(120,4),(121,5)</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>117</td>
<td>(115,1),(116,1),(118,3),(119,4),(120,5),(121,3)</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>118</td>
<td>(115,3),(116,4),(117,3),(119,4),(120,4),(121,4)</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>119</td>
<td>(115,5),(116,2),(117,4),(118,4),(120,11),(121,8)</td>
<td>6</td>
<td>34</td>
</tr>
<tr>
<td>120</td>
<td>(115,6),(116,4),(117,5),(118,4),(119,11),(121,9)</td>
<td>6</td>
<td>39</td>
</tr>
<tr>
<td>121</td>
<td>(115,6),(116,5),(117,3),(118,4),(119,8),(120,9)</td>
<td>6</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 6.6: Conflict Matrix for Dataset 3

shown in Figure 6-11 was generated using longest task first, task ordering heuristic and most available resource heuristic, the number of conflicts generated were 26. All conflicting sessions have been shaded, and it is evident from Figure 6-11 that conflicts have been produced towards the end of the schedule.

Figure 6-12 is an example of an unsuccessful schedule which was generated, in
<table>
<thead>
<tr>
<th>RoomId</th>
<th>Capacity</th>
<th>Available Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm.1</td>
<td>10</td>
<td>(1,15)</td>
</tr>
<tr>
<td>rm.2</td>
<td>10</td>
<td>(1,15)</td>
</tr>
<tr>
<td>rm.3</td>
<td>15</td>
<td>(1,15)</td>
</tr>
<tr>
<td>rm.4</td>
<td>15</td>
<td>(1,15)</td>
</tr>
<tr>
<td>rm.5</td>
<td>15</td>
<td>(1,15)</td>
</tr>
</tbody>
</table>

Table 6.7: Resources Available for Schedule in Dataset 3

<table>
<thead>
<tr>
<th>Id</th>
<th>Sessions Enrolled</th>
<th>Id</th>
<th>Sessions Enrolled</th>
<th>Id</th>
<th>Sessions Enrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[102,103,105]</td>
<td>24</td>
<td>[108,109,110,111]</td>
<td>44</td>
<td>[116,118,121]</td>
</tr>
<tr>
<td>6</td>
<td>[101,102,106]</td>
<td>26</td>
<td>[107,111,114]</td>
<td>46</td>
<td>[117,119,120]</td>
</tr>
<tr>
<td>7</td>
<td>[101,104,107]</td>
<td>27</td>
<td>[107,110,111,112]</td>
<td>47</td>
<td>[119,120,121]</td>
</tr>
<tr>
<td>8</td>
<td>[102,104,106]</td>
<td>28</td>
<td>[107,109,110,112]</td>
<td>48</td>
<td>[115,116,121]</td>
</tr>
</tbody>
</table>

Table 6.8: Enrollment Data for Dataset 3

this case tasks with session identifiers 104, 109, 113, and 119 could not be scheduled. The scheduler is designed in such a manner that it stops the scheduling process once it encounters the first task which is unschedulable which was task 104. Task 104 was unschedulable because its enrollment is 12 and the only resources which can schedule it are Rm.5 and Rm.4 both of which do not have sufficient number of hours. Other resources namely Rm.2 and Rm.3 have sufficient number of hours but not the right capacity to handle the session.
<table>
<thead>
<tr>
<th>Id</th>
<th>Dur</th>
<th>EST</th>
<th>LET</th>
<th>N.ENR</th>
<th>Conflicting Sessions</th>
<th>Total Conflicts</th>
<th>Start</th>
<th>End</th>
<th>Schedulable Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>6</td>
<td>22</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>6</td>
<td>23</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>103</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>6</td>
<td>23</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>104</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>32</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>34</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>106</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>10</td>
<td>35</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>107</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>14</td>
<td>12</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>108</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>9</td>
<td>26</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>109</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>28</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>110</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>10</td>
<td>9</td>
<td>30</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>42</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>113</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>7</td>
<td>9</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>114</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>11</td>
<td>8</td>
<td>27</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>115</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>9</td>
<td>6</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>116</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>6</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>117</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>6</td>
<td>6</td>
<td>17</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>118</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>8</td>
<td>6</td>
<td>22</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>119</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>6</td>
<td>34</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>120</td>
<td>3</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td>6</td>
<td>39</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
<tr>
<td>121</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>6</td>
<td>35</td>
<td>-</td>
<td>-</td>
<td>(1,15)</td>
</tr>
</tbody>
</table>

Table 6.9: Tasks with All Fields - Dataset 3

We will describe the effect of each of the following task ordering heuristics and its interaction with given resource ordering heuristics to see how different schedule heuristics generate different type of schedules. *Infeasibility* in the schedule is mostly produced by two factors either of which can occur or a combination of both. The first factor is *unavailability of rooms with the required capacity* to schedule a task although the resources with lesser capacity may have schedulable hours. This was the reason why in Figure 6-13 a complete schedule could not be generated. The second reason for infeasibility arises when *no resource has enough hours to schedule* the next task. The second situation often arises when tasks are not allocated contiguously to resources and many time gaps occur between tasks for any given resource. So in effect the resource cannot be used to schedule a large task although it may be idle for a time period greater than the length of the task.
Dataset 3: Schedule.

Total Schedule Conflicts = 26.

Task Ordering Heuristics: Longest Task First

Figure 6-11: A Sample Timetabling Schedule

6.9.2 Scheduling using Shortest Task First
The shortest task/session first heuristic is the first one tried out on all the datasets. The shortest task first heuristics did not prove to be a particularly effective heuristic
Dataset 3: Sample Infeasible Schedule

Task Ordering Heuristics: Shortest Task First
Tasks 104, 109, 113, 119 have not been scheduled, scheduler stopped after being unable to schedule 104.

Figure 6-12: An Unsuccessful Assignment of Resources to Tasks

in timetabling scheduling. This is noteworthy since the shortest task first was one of the better task scheduling heuristics in the case of job shop scheduling where it produced schedules with lower makespans and costs.
The plot of schedule conflicts vs schedule iterations for the shortest tasks scheduled first task heuristics for dataset 1 is shown in Figure 6-14. The plot using the same heuristics for dataset 2 and 3 appear in the top half of Figure 6-16 and Figure 6-18 respectively.

The range of conflicts when this heuristic is used in conjunction with closest fitting resource first is between 38-80 for dataset 1, for dataset 2 it is between 40-80 and for dataset 3 it is between 21-34. When the current task heuristic is used with earliest schedulable resource first the range of conflicts are 23-80 for dataset 1, 25-80 for dataset 2 and 21-50 for dataset 3. When the current task heuristic is used with most available resource first the range of conflicts is 42-80 for dataset 1, 20-80 for dataset 2 and no feasible schedule for dataset 3.

The shortest task first heuristic performed best when used in conjunction with the earliest schedulable resource first. In dataset 1 a schedule having 80 conflicts is used to denote an infeasible schedule, and this produced only 2 infeasible schedules in 50 iterations compared to 11-12 infeasible solutions produced in solutions which use the most available resource heuristic within the same number of iterations. The closest fitting room heuristic produced the least number of infeasible solutions as it allocates the most optimal resource with respect to enrollment.

For dataset 3 this schedule fails to produce any schedule in the first 50 iterations, this is because all the conflicting small tasks are scheduled first leaving lots of holes in the schedule. The longer tasks then could not be scheduled. The phenomenon which causes a large amount of infeasible solutions is shown in Figure 6-13.

### 6.9.3 Scheduling using Longest Task First

The longest task first proved to be a better heuristic than the shortest task first heuristic. The plot of schedule conflicts vs schedule iterations for the shortest tasks scheduled first task heuristics for dataset 1 is shown in Figure 6-14. The plot using the same heuristics for dataset 2 and 3 appear in the bottom half of Figure 6-16 and Figure 6-18 respectively.

The range of conflicts when this heuristic is used in conjunction with closest fitting resource first is between 39-54 for dataset 1, for dataset 2 it is between 36-43
Figure 6-13: Why Shortest Task First/Most Avail Resource Generates Infeasible Solutions

and for dataset 3 it is between 28–32. When the current task heuristic is used with earliest schedulable resource first the range of conflicts are 36-51 for dataset 1, 34-46 for dataset 2 and 26-35 for dataset 3. When the current task heuristic is used with most available resource first the range of conflicts is 34-48 for dataset 1, 39-46 for dataset 2 and 25-33 for dataset 3.

In general the scheduling longest tasks first was better especially when the longer tasks had high enrollment and a large amount of conflicts. This outperformed the shortest task first heuristic with any resource heuristic. The longest task first heuristic also produced more compact schedules, where resources have less idle time between consecutive tasks.

6.9.4 Scheduling using Most Conflicting Task First
Task ordering based on most conflicting task first also works on the idea that it is better to schedule more difficult tasks first and it will be easier to schedule less conflicting tasks later. This heuristic performed very well when there were many sessions with high conflicts. The best schedules were obtained when this heuristic
was used in conjunction with closest fitting room first resource heuristic.

The plot of schedule conflicts vs schedule iterations for the shortest tasks scheduled first task heuristics for dataset 1 is shown in Figure 6-15. The plot using the same heuristics for dataset 2 and 3 appear in the top half of Figure 6-17 and Figure 6-19 respectively.

The range of conflicts when this heuristic is used in conjunction with closest fitting resource first is between 30-42 for dataset 1 as shown in Figure 6-15, for dataset 2 it is between 36-43 and for dataset 3 it is between 12-25. When the current task heuristic is used with earliest schedulable resource first the range of conflicts is 30-41 for dataset 1, 33-44 for dataset 2 and 14-28 for dataset 3. When the current task heuristic is used with most available resource first the range of conflicts was much higher and a few infeasible solutions were produced in almost all cases.

6.9.5 Scheduling using Least Schedulable Task First

Again scheduling the least schedulable task first is an attempt to solve the most difficult subproblem of the entire problem, where tasks are dynamically re-arranged so that the least schedulable ones are scheduled first. The performance of this heuristic is quite dependent on the conflict graph of the underlying problem. This heuristic does not produce many different types of schedules if some sessions conflict with a large amount of other sessions as in the case of dataset 1 and 2. The sessions 101, 103, 104, 117, 120 conflict with almost every other session and once these are scheduled all other sessions have very few intervals left where they can be scheduled without conflicts, so all task become equally schedulable, and the same ordering of tasks is produced during the scheduling process. Since Dataset 3 has a two component conflict graph, and only some sessions have high conflicts, this heuristic does produce different results during different schedule permutations.

The plot of schedule conflicts vs schedule iterations for the shortest tasks scheduled first task heuristics for dataset 1 is shown in Figure 6-15. The plot using the same heuristics for dataset 2 and 3 appear in the lower half of Figure 6-17 and Figure 6-19 respectively. As can be seen from the Figure 6-15 both of the resource
ordering heuristics earliest schedulable resource first and closest fitting resource first produce the same result each of the 50 times the iterations were done, and the number of conflicts were 39. Most available resource first produced 44 conflicts each time. Dataset 3 shows a variation and earliest schedulable resource first performed best with conflicts in the range of 16-25. This is very good compared to the performance obtained by scheduling using most available resource first which produced schedules 5-6 infeasible schedules within the same number of iterations and in general produced schedules with 30 conflicts.

6.10 Conclusion

From the simulations conducted it was clear that certain task characteristics and resource characteristics played an important role in determining the nature of schedule which was generated. The exact heuristics which should be used depends on the problem itself, and one cannot state a specific recipe which will work every time to give the best solutions. The guidelines given below are helpful for people developing timetabling schedulers. The following task and resource characteristics should be taken into account while generating any schedule.

- Session Length
- Session Enrollment
- Session Conflicts

For resources the most important characteristics are

- Venue Capacity
- Venue Availability

The length of the session is important and it effects the schedule generated because it is better to schedule longer conflicting tasks first and create small holes in the schedule initially than scheduling smaller conflicting tasks first and leaving holes in the schedule which may not be filled by larger tasks later. When there are resource limitations then it is very important to schedule resources which have a
capacity almost equal to the enrollment of the task. This often leads to scheduling of sessions so that there is less wastage of resource capacity and produces less infeasible schedules. It is better to schedule sessions with higher enrollment first if any other resource heuristics other than closest fitting resource is being used. Scheduling sessions with a higher number of conflicts first also leads to better schedules especially when the conflicting sessions have large enrollment and are of longer duration. Scheduling these difficult tasks first generally leads to schedules with lesser number of conflicts and also lesser number of infeasible schedules, some exceptions do occur if the most conflicting tasks are all very small, or have a very low enrollment. The user can furthermore experiment with a combination of these strategies by defining his/her own heuristic depending upon the situation.
Figure 6.14: Schedule Conflict Plot using Shortest, Longest Task First (Dataset1)
Figure 6.15: Schedule Conflict Plot using Most Conflicting, Least Schedulable Task (Dataset1)
Figure 6.16: Schedule Conflict Plot using Shortest, Longest Task First (Dataset2)
Figure 6.17: Schedule Conflict Plot using Most Conflicting, Least Schedulable Task (Dataset2)
Figure 6.18: Schedule Conflict Plot using Shortest, Longest Task First (Dataset3)
Figure 6.19: Schedule Conflict Plot using Most Conflicting, Least Schedulable Task (Dataset3)
Chapter 7

Process Schedulers

"Real-world problems are often "high-dimensional", that is, are described by large numbers of dependent variables. Algorithms must be specifically designed to function well in such high-dimensional spaces."

-David Rogers

- Weather Prediction Using a Genetic Memory

7.1 Introduction

In this chapter titled "Process schedulers" we sketch how scheduling can be done in a process plant. Process plant scheduling is not reported widely in the OR literature, and the basic ideas expressed in this chapter were in response to solving a real life problem which arose in the petroleum industry. The methods developed here can be used to model any process plant scheduling problem, which is essentially a flow problem consisting of solid, liquid or gas flow, unit operations like heat transfer, mass transfer and finally the flow of the final product out.

Some of the scheduling problems which commonly occur in the process plant scheduling cluster are:

Petroleum Blending Scheduling: The blending problem consists of a petroleum
refinery, which produces different grades of petroleum products; the task addressed is to determine the production schedule given end product requirements. The refinery refines crude oil in a series of operations. A simplistic scenario involves breaking down the crude into fractional distillates; these distillates are blended to produce end products whose quality depend on the constituent components as well as on the time of blending. A blending scheduler schedules the blending operations of different constituents to produce end products shipped out of the refinery.

Chemical Plant Scheduling: Other chemical plants require scheduling the production of chemicals like acetylene, methanol. Raw materials - mainly hydrocarbons are mixed in different chambers and are fed into a catalytic chamber/vessel where chemical reactions take place. Very often the same plant is used for scheduling a wide variety of chemical products, and the demand for various products depends on the particular day. In this case the scheduling operation will determine the start and end times of each unit operation in the chemical plant as the same resources will be used for more than one product. Other types of restrictions may require a particular sequencing of products which in the sense of a jobshop scheduler is like precedence constraints.

7.2 Problem Statement

The input to the process scheduler consists of the following items

Product List: This is a list of products to be shipped through the external pipeline station.

Blending Recipes with individual costs: Possible blending recipes of all products with individual component costs in a recipe are given. This information could alternately be supplied as a linear programming formulation, the LP objective being to minimize the cost of products subject to a set of constraints (generally of the $\geq$ type). These constraints ensure a minimum quality of each product.

Pipeline Schedule: The pipeline schedule of the end products involves the specification of the time window in which the products need to be shipped.
Structural Constraints: These involve the physical refinery layout. Tank layout places restrictions on the blending as it restricts the flow of fluids from storage to maybe exclusive periods if a collection of tanks have a common outlet (the common suction constraint); also they place a restriction on the source and destination of intermediate components and products.

Some of these are specified as explicit inputs to the problem, for example the quantities and time schedules of specific end products are mentioned explicitly with each problem; while other inputs are stored as predicates in the program. A sample plant layout is presented in Figure 7-1.

7.3 Blending Scheduler Details
The Blending Scheduler problem is a specific instance of the general process scheduling problem. The basic components of a generic scheduler are tasks, resources and constraints. The scheduling operations here can be mapped to schedule different fluid streams at various parts of the plant layout. The resources are the plant infrastructure consisting of component tanks, the reaction tanks, storage tanks, and pipelines. The constraints consist of plant layout constraints, sequencing constraints and deadline constraints. The generic scheduler takes as input the task, resource, constraint specification and produces as output the schedule for each product specified in the task list.

<table>
<thead>
<tr>
<th>Product</th>
<th>Min Quantity (gal)</th>
<th>Max Quantity (gal)</th>
<th>Earliest Time Wanted (W1)</th>
<th>Latest Time Wanted (W2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Pr_1$</td>
<td>100</td>
<td>150</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>$Pr_2$</td>
<td>30</td>
<td>50</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$Pr_n$</td>
<td>100</td>
<td>120</td>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 7.1: Example Tasks for a Process Plant

7.3.1 Tasks
The tasks in the system are to schedule a list of product streams between given durations. A sample task input is shown in Table 7.1. The task fields are id the product identifier, min_quant, max_quant which are the minimum and maximum
volume of product needed, W1 and W2 which are the time window boundaries between which the product must be delivered at the output station.

Using the object representation scheme which has been developed earlier in this thesis, the tasks in the system can be represented as follows:
7.3.2 Resources

The three types of resources in the system are tanks, reactor vessels and pipelines. Other resources assumed implicitly here are valves and pumps, we will not go into the details of pump ratings and valve capacities. Tanks which store raw material are input tanks, other tanks are used to hold the final product which is ready to be shipped outside. The reactor vessel as the name suggests is a place where reactions occur and has inlets from other tanks. Most reactions take a specific amount of time, after which the product is pumped out of the reactor vessel. The resulting product can be pumped directly out of the process plant or kept in an intermediate storage tank from where it is pumped out at a later time.

Tanks are connected according to the connectivity plan of the plant. The capacity (flowrate) of different pipelines is fixed, so time taken to fill a tank is completely specified by the capacity of the tank and capacity of the flowpipe leading to it. Table 7.2 illustrates some of the features of tanks in a typical plant layout.

<table>
<thead>
<tr>
<th>Tank Id</th>
<th>Component Type</th>
<th>Content Volume</th>
<th>Maximum Capacity</th>
<th>In Connect</th>
<th>Out Connect</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>c1</td>
<td>200</td>
<td>500</td>
<td>[P1]</td>
<td>[R1,R2]</td>
</tr>
<tr>
<td>T2</td>
<td>c2</td>
<td>300</td>
<td>600</td>
<td>[P2]</td>
<td>[R2,R4]</td>
</tr>
<tr>
<td>T3</td>
<td>c3</td>
<td>400</td>
<td>400</td>
<td>[P3]</td>
<td>[R1,R2]</td>
</tr>
<tr>
<td>T4</td>
<td>c4</td>
<td>300</td>
<td>600</td>
<td>[P3]</td>
<td>[R3,R4]</td>
</tr>
<tr>
<td>T5</td>
<td>empty</td>
<td>500</td>
<td>[P3]</td>
<td></td>
<td>[R2,R3]</td>
</tr>
<tr>
<td>T6</td>
<td>empty</td>
<td>500</td>
<td>[P3]</td>
<td></td>
<td>[R1]</td>
</tr>
<tr>
<td>T7</td>
<td>empty</td>
<td>450</td>
<td>[R1,R2]</td>
<td></td>
<td>[P4]</td>
</tr>
<tr>
<td>T8</td>
<td>empty</td>
<td>500</td>
<td>[R3]</td>
<td></td>
<td>[P5]</td>
</tr>
<tr>
<td>T9</td>
<td>empty</td>
<td>500</td>
<td>[R3]</td>
<td></td>
<td>[P6]</td>
</tr>
</tbody>
</table>

Table 7.2: Example Tank Specification in a Plant

Using the object representation scheme which has been developed earlier in this thesis the resources in the system can be represented as follows:
<table>
<thead>
<tr>
<th>Reactor</th>
<th>Max Cap (gal)</th>
<th>In Connect</th>
<th>Out Connect</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>600</td>
<td>[T1, T3]</td>
<td>[P1, T6, T7, R4]</td>
</tr>
<tr>
<td>R2</td>
<td>600</td>
<td>[T1, T2, T3]</td>
<td>[T7, P5]</td>
</tr>
<tr>
<td>R3</td>
<td>600</td>
<td>[T4, T5]</td>
<td>[P5, T8, T9]</td>
</tr>
<tr>
<td>R4</td>
<td>600</td>
<td>[T2, T4]</td>
<td>[P4, T7]</td>
</tr>
</tbody>
</table>

Table 7.3: Reaction Vessels in the Plant

rcs([id: t1, typ: tank, content: c1, vol: 200, maxcap: 500, in: [P1], out: [R1, R2]]).
rcs([id: t2, typ: tank, content: c2, vol: 300, maxcap: 600, in: [P2], out: [R2, R4]]).
rcs([id: t3, typ: tank, content: c3, vol: 400, maxcap: 400, in: [P3], out: [R1, R2]]).
rcs([id: t4, typ: tank, content: c4, vol: 300, maxcap: 600, in: [P3], out: [R3, R4]]).
rcs([id: r1, typ: tank, content: nil, vol: 0, maxcap: 600, in: [T1, T3], out: [P1, T6, T7, R4]]).
rcs([id: r2, typ: reactor, content: nil, vol: 0, maxcap: 600, in: [T1, T2, T3], out: [T7, P5]])

External Pipelines

It is required to meet the piping schedule of a number of pipes \( P_1 \ldots P_n \). Most plants have an exclusive pumping station from where a set of pipelines go into the plant thus carrying raw materials into the plant. These pipes are connected from a piping station to external destinations. We assume that pipelines \( P_1 \) through \( P_n \) are completely independent.

7.3.3 Blending Recipes

A blending recipe consists of the specification of a product, along with a combination of components which can be blended to provide the final product.

Table 7.4 is an example table which illustrates a set of blending recipes considered for products \( P_{r1} \) through \( P_{r5} \). c1 to c5 are component numbers while \( P_{r1} \) to \( P_{r5} \) are the 5 products which need to be shipped through the pipeline.

In Prolog the recipes are represented as shown next. Each blend recipe consists of a product name, followed by a list of blending component/reaction times. For example product pr1 has two recipes in the clause represented. The first one is a mixture of components c1, c2 in the ratio of 5:3. The reaction time for this mixture is 2. This is followed by mixing the resulting compound and c3 in a 2 part ratio and
<table>
<thead>
<tr>
<th>Product</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>Time</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Pr_1$</td>
<td>0.5</td>
<td>0.3</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
<td>2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>$Pr_2$</td>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
<td>0.2</td>
<td></td>
<td>4</td>
<td></td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>2</td>
</tr>
<tr>
<td>$Pr_3$</td>
<td>0.3</td>
<td>0.2</td>
<td>0.4</td>
<td>0.4</td>
<td></td>
<td>2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>$Pr_4$</td>
<td>0.5</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td></td>
<td>3</td>
<td></td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>$Pr_5$</td>
<td>0.2</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
<td></td>
<td>4</td>
<td></td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.4: Sample Blend Recipes

holding the resulting mixture for 2 time units. The second recipe calls for mixing components $c_1$, $c_2$ and $c_4$ in a 2:3:5 ratio with a reaction time of 5.

blend_recipe(pr1,[[[(c1,0.5),(c2,0.3)]-2,((c3,0.2))]-2],[[((c1,0.2),(c2,0.3),(c4,0.5))]-5])

blend_recipe(pr2,[[[(c1,0.2),(c2,0.8)]-4],[[((c2,0.1),(c3,0.2))]-1],[[((c5,0.7))]-2]])

blend_recipe(pr3,[[[(c2,0.5),(c2,0.4)]-2,((c1,0.1))-3],[[((c1,0.3),(c2,0.2))]-2,[[((c5,0.5))]-2]]]

7.4 Constraints

Constraints are specified implicitly in the previous description. There are some additional constraints which must be met, these are - structural constraints, product deadline constraints, product sequence constraints. Product composition constraints also determine the choice of starting components.

Structural Constraints: The structural constraints manifest themselves in terms of physical connectivity of the system. Certain tanks share a common suction, hence their product can only be drained in a mutually exclusive fashion. Similarly certain tanks at the input share a common suction, so they can be filled only mutually exclusively.

Product Deadline Constraints: Although a global time window of operation is specified for the entire product shipment schedule, products have to be shipped within a specified product window. eg. Product $Pr_1$ needs to be
shipped during the scheduling window 10-14, where 10 is the earliest time and
14 is the latest time before which \( Pr_1 \) must be shipped.

**Product Sequence Constraints:** Certain products in the pipeline affect com-
position of other products which would follow in the pipeline. Hence there is
a constraint on the sequence of products in the pipeline. eg. \( Pr_3 \) must follow
\( Pr_4 \) if they are shipped on the same pipe.

### 7.5 Process Scheduling Example

We now present the sequence of scheduling in a small example to illustrate how
process scheduling works. The plant layout used is presented in Figure 7-1, there
are 9 tanks \( T_1, \ldots, T_9 \) and 4 reactors \( R_1, \ldots, R_4 \). Pipelines \( P_1, \ldots, P_3 \) are input
pipelines using which input raw material is pumped into tanks \( T_1, T_2, T_3, T_4 \) and
\( T_5 \). The schedule developed is presented in Figures 7-2 and 7-3. Three products
\( Pr_1, Pr_2 \) and product \( Pr_3 \) have to be made, the exact amounts and time between
which the products must be delivered is presented in Figure 7-2.

**Product \( Pr_1 \):** Product \( Pr_1 \) is the first product chosen to be made. The blending
recipe used consists of mixing 50 parts of component \( C_1 \) and 30 parts of
component \( C_2 \), followed by 20 parts of \( C_3 \). The first reaction takes 2 units of
time and is to be followed by mixing 20 parts of component \( C_3 \) to the reacted
mixture.

Component \( C_1 \) is available in tank \( T_1 \) while component \( C_2 \) is available in tank
\( T_2 \). Reactor \( R_1 \) is used for the first reaction, the reactor is tied up during
three phases, the input phase when mixtures are pumped into the reactor, the
reaction phase when the actual reaction takes place, and lastly the third phase
when material is pumped out of the reactor through a pipeline to reactor \( R_4 \).
After a hold time of 2 time units, which is required by the blending recipe, the
mixture is pumped into reactor \( R_4 \). The gantt chart showing all the resources
used is appears in Figure 7-2.

**Product \( Pr_2 \):** Product \( Pr_2 \) is the second product which is scheduled. The blending
recipe used for making this product uses 20 parts of component \( C_1 \) and 80 parts
Figure 7-2: Process Scheduling Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pr1</td>
<td>100</td>
<td>150</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>Pr2</td>
<td>30</td>
<td>70</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Pr3</td>
<td>50</td>
<td>70</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Updated Resource Status

400

Product Chosen = Pr2

Blend Recipe: 20 c1 (tank t1) + 80 c2 (tank t2) -> Reactor r2 for 4 time units

Figure 7-3: Process Scheduling Example Continued
of component C2. Component C1 is obtained from tank T1 and component C2 is obtained from tank T2. The reaction hold time is 4 units and the reactor vessel used is R2. The gantt chart showing all the resources used is appears in Figure 7-3. The schedule for product P3 is not shown in the figure and is generated in a similar manner.

7.6 Process Scheduler Algorithm

The basic steps used in scheduling in a process plant scheduler is a specific instance of the algorithm presented in Figure 3-11. The schedule cycle remains the same and involves first a task selection based on task selection heuristics. The task selection also involves selecting a suitable blending recipe if one exists. This is followed by choosing a resource which itself is based on a schedule recipe. Once allocation of resource to task is done, it is followed by constraint propagation. The constraint propagation step involves setting the earliest schedulable times of products whose earliest start times are dependent on the schedule times of other products due to precedence constraints.

1. **Task Selection:** Choose a task (Product to be produced). Select a blend recipe which indicates the different components to be blended as well as reaction times.

2. **Resource Selection:** Choose resources based on the blend recipe. The time of availability of each resource depends on resource availability. A collection of resources may be required as several tanks and reactors may be needed to take part in the process.

3. **Allocation and Updating:** Allocate resources to the task. Make the collection of resources unavailable on the gantt chart.

4. **Propagate constraints:** Once a product is scheduled, make all products it precedes schedulable.

5. **Goto Step 1.**

On reaching a deadend the algorithm backtracks chronologically. Backtracking is allowed on the choices at the task selection, blending recipe selection and resource
combination selection. If a particular resource combination cannot be obtained, backtracking is done on blending recipe choice. The next level of backtracking is done at the task choice level.

7.7 Process Scheduler Heuristics

The same kind of heuristics which were used in the jobshop schedulers, transportation schedulers, and timetabling schedulers can be used here. We briefly present some of the task selection and resource allocation heuristics which can be used in the process plant scheduling instance.

Task selection heuristics:

Some of the common task selection heuristics used in the case of process schedulers is presented in this section. All heuristics arrange tasks based on the value of one of their attributes. The user can also define his/her own heuristic based on one of the attribute values.

Earliest Due Date First:

The tasks are scheduled based on their task attribute \( W_1 \) which is the lower window boundary. This heuristic has been used in all other schedule clusters, and the basic idea is to schedule more difficult tasks first. Early due dates is a measure of difficulty, the earlier a task is due the more difficult it is. However each task has a schedulable window \( W_1 \) and \( W_2 \) and tasks could have a very low value of \( W_1 \) and a very high value of \( W_2 \), so scheduling tasks with smallest value of \( W_1 \) may not always prove to be the best heuristic.

Smallest Schedulable Window First:

In this heuristic tasks having the smallest schedulable window i.e. \( W_2 - W_1 \) are scheduled first. The smaller the schedulable window, the more difficult it is to schedule a task. This is a good heuristic in practice, but this heuristic also has a drawback. The main drawback is that it may be easier to schedule a task with a small schedulable window but a very large \( W_1 \) i.e. lower window limit. In such a scenario the final product can wait in a product tank till the time indicated in its lower schedulable interval.
Highest Quantity First:

In this heuristics, the product whose volume is maximum is scheduled first. The idea here is that the amount of volume of product is also a measure of difficulty and more volume of product means higher resource utilization.

Lowest Quantity First:

If the basic aim of the schedule generated is only to reduce the number of tardy or late jobs, then it as better to schedule those tasks first where the product volume required is less. This is similar to smallest task first heuristic in the case of jobshop scheduling.

Resource selection heuristics:

The resource selection is based on the recipe which is selected. Once the recipe is chosen, resource selection really depends on the availability of raw materials. The only heuristic used in the implemented instance of the process scheduler schedules resources having maximum quantity of the required raw material first.

7.8 Conclusions

In this chapter, a design of a process scheduler has been presented. Details of a particular instance of the process plan scheduler called a blending scheduler were also presented. The basic scheduling strategy for process schedulers remains the same generate and test strategy, presented in this thesis. In fact a process scheduler on first glance looks quite different from a jobshop, timetabling or transportation scheduler, but the final scheduling process boils down to an allocation of tasks to resources. The main challenge is to identify tasks, resources and constraints, which has been done with respect to process scheduling in this chapter. The basic strategy used is to select a particular product to be blended, select a blend recipe and then allocate a group of resources based on the blend recipe. This is followed by updating local data structures involving the availability and quantity of material which is present in a resource. Constraint propagation modifies the schedulable periods of other products based upon the current scheduled product.
On reaching deadends the schedule backtracks over the choice of resources, then the choice of a blending recipe and then the choice of a task. The primary aim of the scheduler may be to minimize the cost of the schedule which depends of the blend recipes used, or to have the minimum number of tardy outputs, the latter was actually the aim of a process scheduler implemented using ideas developed in this chapter.
Chapter 8

Concluding Remarks

"Adde parvum parvo magnus acervus erit."
(Add little to little and there will be a big pile.)
- Ovid

"The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" (I found it!) but "That's funny ..."
- Isaac Asimov

8.1 Contributions

This thesis titled "On Designing an AI Based Generic Scheduling Framework" primarily is a response to a need for systematic development of different schedulers with a quick turnaround time. A Generalized Scheduling Framework/Environment allows a person to develop schedulers in a systematic manner with short turn around times using a semi-automatic development process and gives end users the flexibility to design new heuristics and find out how best to generate schedules subject to specific constraints.

Some of the specific contributions of this research are:

- This thesis advocates the categorization of schedulers into different clusters depending upon problem characteristics. Any specific scheduler instance is
obtained by customizing the base cluster. By using the cluster design concept and a *generate and test scheduling strategy* a scheduler can be designed with a quick turnaround time.

- A group of practical scheduler clusters namely jobshop, transportation, timetabling and process schedulers have been presented, and there is a provision in the framework to extend each of the clusters to take care of specific constraints. The thesis identifies the key entities in any schedule namely tasks, resources and constraints and shows how they can be suitably represented using a Prolog based object representation scheme.

- This thesis presents a unified problem solving strategy for a large group of scheduling problems. A general purpose *generate and test scheduling engine* has been presented in Chapter 3 which can be used for scheduling different applications. The scheduling strategy has been presented in great detail in Figure 3-11 and specialized versions of the algorithm appear in Figures 4-2, 5-3, 6-5, and Section 7.6 for the case of jobshop, transportation, timetabling and process plant schedulers.

- The thesis enables the end user to test the effect of different scheduling strategies on the performance of the system. He/She can build a strategy depending on the scheduling problem at hand, hence the framework provides a good method for developing scheduler prototypes.

### 8.2 Scheduler Clusters

"*Scheduling is the allocation of resources to tasks subject to constraints*”. Assuming that tasks, resources and constraints can be identified for each scheduler, then at a high level of abstraction, all schedulers are the same. Hence the challenge is in clearly identifying and representing tasks, resource and constraints for a scheduler and problem specific concepts unique to each problem domain. Intuitively some scheduling problems are very closely related, for example examination scheduling, conference scheduling and classroom scheduling are more closely related than say the relationship of examination scheduling to truck dispatching. In order to capture
this intuitive notion and also allow a range of schedulers to exist in a generic scheduling framework, we classify schedulers based on their characteristics, and schedulers belonging to the same family are said to belong to a scheduler cluster.

The classification of schedulers that we have developed is shown in Figure 3-2. Some scheduler clusters which have been designed are jobshop, transportation, timetabling and process schedulers. Jobshop schedulers assign resources to tasks subject to constraints such as precedence constraints between tasks, transportation schedulers assign transporters to deliver material from source locations to several destinations, and timetabling schedulers allocate venues to sessions such that the cost of allocation or the number of conflicts in the system are minimized.

All the scheduler clusters have a similar core part, consisting of the basic attributes needed for the schedulers and useful predicates implementing central concepts. Particular schedulers are built by customizing and instantiating the scheduler clusters. For example, the ocean tanker scheduler or truck dispatch scheduler are instances of a transportation scheduler and are built by extensions and customizations done to the transportation scheduler cluster. The representation scheme consists of a Prolog based object representation scheme where the tasks, resources and constraints are expressed.

8.3 On Designing Clusters

As mentioned in the previous section the challenge in defining a new cluster depends on identifying and representing tasks, resource and constraints for a scheduler and also taking care of some problem specific concepts unique to each problem domain. In order to design a new cluster one must first understand the scheduling problem, and identify any key concepts unique to the cluster. Example of concepts unique to a particular scheduler are the routing concept for transportation scheduler, or conflict matrix for timetabling schedulers. The cluster specific concepts help in designing the task, resource, and constraint fields as well as in performing cluster specific calculations and evaluating the quality of generated schedules. A scheduling cluster may not have a unique problem characteristic, in which case one can immediately design tasks and resource attribute fields.
The next important thing is to design the fields for tasks, resources and constraints. Constraints may also be represented as facts. For example precedence constraints for jobshop schedulers were represented as precedes( Task, [list of Tasks it Precedes] ). Each task object needs to have an identifier, name of task, and a scheduling interval between which it must be completed. Very often the interval is represented as a window $W1-W2$. A resource object will contain a resource identifier, the capacity of the resource and time of availability again represented as a window $W1-W2$. It could have additional fields based on the nature of the problem.

The Prolog object system and time handling system described in Chapter 3 are general purpose and can be used in all new clusters developed together with a customized version of the generic generate and test scheduling algorithm described in Chapter 3.

8.4 Schedule Generation

The algorithm presented in Figure 3-11 gives the essence of the schedulers developed under the generalized scheduler framework. The actual implementations of the algorithm are customized for specific schedulers with minor variations. The core of the scheduler consists of a task dispatcher, resource allocator and constraint propagator. Control of the scheduler is provided by the heuristics selector which guides the schedule by determining the task selection and resource selection heuristics. We review each in turn.

**Task Dispatcher:** This determines the order in which different tasks are dispatched in the system. Before the scheduler starts, the tasks are ordered according to the specified task ordering heuristics. As the scheduler runs, tasks are selected for dispatch according to this order. The task dispatcher makes sure that each dispatched task satisfies all temporal constraints before being passed to the resource allocator.

**Resource Allocator:** The resource allocator allocates resources to a schedulable task. Since a number of resource-task allocation combinations exist for each task, the resource allocator is guided by resource selection heuristics to determine resource selection for each task. The resource allocator also determines
the start time of each task, as it depends both on the temporal constraints on
the task, and the periods of availability of the resource.

**Constraint Propagator:** Once a task is allocated resources and has been placed
along the time line, the constraint propagator propagates constraints in the
system. The constraint propagator determines the next set of tasks which
contend for dispatch, and the earliest start times are determined by the con-
straints in the system.

**Heuristics Selector:** The heuristics selector provides flow of control information
to the task dispatcher and resource allocator units. The heuristics selector
can use a user-defined heuristic, or one from the set of standard heuristics
defined for each scheduling cluster in the system. Task dispatch ordering is
determined by the task selection heuristic, while resource allocation is guided
by using resource selection heuristics. Pre-defined task selection heuristics in
jobshop schedulers associate dispatch weights with all the tasks which need
to be scheduled. In the transportation schedulers tasks are sorted based on
one of the attributes of the task object. Resource ordering heuristics also
order resources by one of their attributes and the task and resource ordering
is updated at the end of each schedule iteration.

### 8.5 Closing Remarks

The Generalized Scheduling Framework/Environment described in this thesis allows
a person to develop schedulers in a systematic manner with short turn around
times using a semi-automatic development process and gives end users the flexibility
to design new heuristics and find out how best to generate schedules subject to
specific constraints. We believe that the “AI Based Generic Scheduling Framework”
described in this thesis will prove to be useful to scheduler developers and end users.
Bibliography


[Jac84] Robert Jacobs. OPT Uncovered: Many Production Planning and Scheduling Concepts can be applied with or without the Software. Industrial Engineering, 16(10):32–41, October 1984.


