THE EARLY-TARDY DISTINCT DUE DATE MACHINE SCHEDULING PROBLEM WITH JOB SPLITTING

BERNICE HO KAH YUAN

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgements

In the course of this project, I have replied upon the help of several people to get me through this time.

To my friends Ang Juay Chin and Ho Wee Kit, who were given the task of creating an interface for an unusual scheduling problem by our supervisor at that time, A/P Andrew Lim. It was listening to the problem specifications given to them that first piqued my interest in the project, and they have my apologies if I stepped on their toes at that time with my nuisance questions.

Also, to my family, who gave me moral support throughout the project, even if they never seemed to have any idea what I was talking about most of the time.

Last but not least, to my husband Oon Wee Chong, who has given me loving support over the years, encouraged me and scolded me when I needed it, and put up with my temperaments and almost-daily hay fever attacks.

Table of Contents

Page No.

Summary

In this thesis, we introduce a new practical scheduling problem, the early-tardy machine scheduling problem with distinct due dates and splitting jobs with setup times on uniform parallel machines (ETDDDsplit). It combines characteristics from 2 known scheduling problems - the early-tardy scheduling problem with distinct due dates (on a single machine or on parallel machines) as well as the relatively lesser known scheduling problem with splitting jobs (also called "lotsizing"). In early-tardy scheduling, the aim is to minimize the total weighted distance between each job's deadline and actual completion time. The weights are denoted by early and tardy penalties. In scheduling with job splitting, scheduling with preemption is taken a step further, such that a job's subsections can be simultaneously scheduled on different parallel machines. The ETDDDsplit problem is NP-complete, and does not appear in any literature to date.

We applied (singly and in combination) the standard search techniques of Tabu search, genetic algorithm and simulated annealing, along with a new greedy heuristic, to variations of the problem. The heuristics were run on test cases created using 200 jobs on 5 parallel machines of varying processing rates, to be scheduled over a maximum period of 2000 time units. The jobs had machine- and sequence-independent setup times and random processing requirements. Besides the standard set of jobs, 2 additional sets of special jobs were included - maintenance jobs and breakdown jobs. In cases where the jobs were well-spread out over the entire scheduled period, the heuristics generally performed well. However, as the jobs' release dates were placed closer and closer together, the heuristics' performances deteriorated. By comparing the heuristics' performances, it was inferred that the solution topology of this problem is very flat, making it very difficult to find the global optimal solution.

There is much room for further research in many aspects of this problem, such as a more accurate modeling of the problem and refinements of the search techniques applied to the problem.

List of Figures and Tables

Chapter 1

Introduction and Outline

As long as there are limited resources to achieve our aims, scheduling problems will always exist in various shapes and forms, both in our personal and professional lives. Finding the optimal way to apply our finite resources to the tasks at hand has been the focal point of much research not only in the field of computer science, but also in mathematics, operations research and engineering. The fact that so many scheduling problems are NP-complete [Garey and Johnson, 1979] means that we will probably be wrestling with this demon for many years to come. The long title of this thesis reflects the research that has been already been done on scheduling problems, and hence the extensive specifications needed to describe a unique scheduling problem.

Out of this multitude of possible scheduling problems, why did I decide to work on this particular problem? It was happenstance that I overheard others' work which piqued my interest. Some friends needed to create an interface for a T-shirt manufactory which had a non-standard scheduling problem. It incorporated some facets of a few known problems (specifically, the early-tardy machine scheduling problem and the machine scheduling problem with splitting jobs), but was different enough from all of them that a different approach to solving it needed to be considered. At that time, I was still casting about looking for a research topic that interested me, and started to investigate it. It was to my surprise that I could not find any prior research on this particular problem, despite seeming to be such a reasonable set of restrictions under real-life conditions. This work is merely starts to examine this problem, and I hope that more work can eventually be done so that good (if not optimal) solutions can be consistently generated for it.

In this thesis, we examine a new machine scheduling problem with practical applications, and look at some heuristics that can be applied to generate solutions under varying conditions. In Chapter 2, we first look at the standard scheduling models and notation that is commonly used. In Chapter 3, we fully describe the ETDDDsplit problem together with other scheduling problems it can be related to. In Chapter 4, we review some of the standard scheduling techniques used on problems similar to the ETDDDsplit problem. In Chapter 5, we focus our attention on the heuristics that were used to tackle the ETDDDsplit problem and show that it is indeed NP-complete. Chapter 6 outlines some tests run to determine what values should be used for each heuristic, then describes the test data and heuristic variables used for the experiments, ending with a discussion on the results. In Chapter 7, we conclude with our general findings and thoughts on possible future directions for research on this problem.

Chapter 2

Scheduling Models and Notation

Scheduling problems involve the optimal allocation of resources to activities (jobs) over time. There are many scheduling models to cover a wide range of real-life and theoretical scheduling problems. Usually, each model can be defined by 3 parameters – the machine environment, the optimality criteria and the job characteristics and constraints. This is represented in the standard notation - a 3-field classification system α / β / γ as given by numerous authors [Graham et al, 1979; Lawler et al, 1993; Chen et al, 1998] where :

α denotes the machine environment and number of machines, $\alpha \in \{1, P, Q, R, O, F, J\}$. If we define the machine processing rate to be r_{ij} (the amount of job J_j that M_i processes in 1 time unit), a machine M_i processing the whole of job J_j with a setup time s_j would take $(p_j/r_{ij} + s_j)$ time units to finish processing the job.

- $\alpha = 1$ (a single machine with machine rate r)

In problems where more than 1 machine is present, there can be 2 types of machines - parallel (where any machine can perform any task) and dedicated (where machines are specialized for different tasks). For parallel machines, the problem categories are :

- $\alpha = P$ (identical parallel machines, $p_j/r_{ij} = p_j/r$, where r is independent of job and machine)
- $\alpha = Q$ (uniform parallel machines, $p_j/r_{ij} = p_j/r_i$, where r_i is machine-dependent)
- $\alpha = R$ (unrelated parallel machines where r_{ij} is both job- and machine-dependent)

In a shop environment with dedicated machines, each job has up to *m* operations. Each operation must be processed on a different specific machine and may be of varying lengths. For dedicated machines, the problem categories are :

- α = *O* (open shop : the operations of a job can be processed in any order as long as no operations of the same job are processed simultaneously)
- α = *J* (job shop : there exists a total ordering on the operations of a job, and an operation may not commence until its predecessor is complete)
- α = *F* (flow shop : similar to job shop, but each job's operations have the same total order with respect to machine usage, though operation lengths may differ)

In all standard cases, each machine can process only 1 job (or its operations) at a time. If the problem has a fixed number of machines, that number is included in the environment specification.

β denotes various constraints and job characteristics. This relates to the following characteristics:

- whether jobs have release dates
- the relationship (if any) between release dates, due dates and processing time
- the nature of the jobs' due dates (e.g. common due dates "CDD", distinct due dates "DDD", common due windows, the due date assignment problem)
- whether preemption is allowed (i.e. an job being processed can be interrupted before completion and then continued later)

- whether precedence constraints exist (i.e. a partial order on the set of jobs is given such that $J_i \prec J_j$ implies J_i must be completed before J_j can begin being processed)
- the relationship (if any) between processing times and weights of each job
- whether machines have an idle time restriction
- whether the family scheduling model is used (also indicating the dependencies of the inter-family setup times, batch- or job-availability assumptions and batching machine assumptions if the family model is used)

Almost all research has assumed that each job can be processed on at most 1 machine at a time. However, Xing and Zhang [Xing and Zhang, 2000] referred to an additional characteristic specific to parallel machine scheduling,

- whether each job can be processed on more than 1 machine at the same time. We thus differentiate "splitting" (where each job can have subsections that are processed on more than 1 machine simultaneously) from "preemption" (where jobs can be broken into subsections and processed at non-overlapping times).

γ denotes the optimality criterion used to measure how good any given schedule is. Some common criteria used are $\gamma \in \{C_{max}, \max L_j, \sum C_j, \sum U_j, \sum T_j, \sum w_j C_j, \sum w_j U_j\}$. A more detailed description of all the fields' parameters can be found in Finke et al's bibliographic review [Finke et al, 2002].

Standard scheduling problems can usually be described as m machines M_i ($i = 1, \dots, m$) being available to process n jobs J_j ($j = 1, \dots, n$). Each scheduled job J_j may have the following characteristics :

- processing requirement p_j (size of the job, the actual processing time required depends on the rate of the machine that is processing the job J_j)
- due date or deadline d_j
- setup time s_{kj} ("idle time" required on a machine after processing J_k before processing J_j, considered sequence-dependent if $s_{kj} \neq s_{jk}$)
- release date a_i (the earliest time the job can start being processed)
- actual completion time C_j
- lateness $L_j = c_j - d_j$
- tardiness $T_j = \max\{0, L_j\}$
- unit penalty U_j ($U_j = 0$ if $c_j < d_j$, $U_j = 1$ otherwise)
- weight w_j (a measure of the importance of a job)

Instead of each job having an individual due date, another measure of the "goodness" of a schedule is :

- makespan or maximum completion time of all the jobs, C_{max}

James and Buchanan [James and Buchanan, 1997] used an alternative to w_j which was replaced by 2 parameters for each job. These parameters are used when trying to minimize the total weighted distance between the due date and completion date :

- early penalty e_j (cost per unit time for early completion i.e. $C_j < d_j$)
- tardy penalty t_j (cost per unit time for late completion i.e. $C_j > d_j$)
    - using this, the penalty for a job J_j would be $\begin{cases} e_j|L_j| & (C_j < d_j) \\ t_j|L_j| & (C_j > d_j) \\ 0 & (C_j = d_j) \end{cases}$

A completed schedule then describes for each job a set of time-units on specific machines that fulfills its processing requirements whilst satisfying the problem's other constraints.

Chapter 3

Problem Description

Before describing the problem that is the focus of this thesis, we first look at some related scheduling problems that are quite closely related.

3.1 Related problems

3.1.1 Early-Tardy Machine Scheduling

The general early-tardy (ET) machine scheduling problem closely models the Just-In-Time logistical requirements of many industries today, since there are not only often penalties for completing a job too late, but also penalties for completing a job too early. ET scheduling problems in general are NP-complete [Garey et al, 1978]. There has been much research conducted on the common due date (CDD) scheduling problem and its variations, although preemption is usually not included. These include problems with job setup time requirements [Rabadi et al, 2002] and multiple parallel identical machines [Emmons, 1987; Kubiak et al, 1990].

A more general model of the ET scheduling problems uses distinct due dates (DDD). Research in this area includes work on variations where the problem

- has the same early and late penalties for all jobs on a single machine [Fry et al, 1987]
- has early and late penalties proportional to the processing time of each job on a single machine [Yano and Kim, 1991]

- has unrelated early and late penalties on a single machine [James and Buchanan, 1997]
- does not permit any idle machine times, with a single machine [Ow and Morton, 1989]
- has parallel identical machines [Hamad et al, 2002]

Baker and Scudder's review [Baker and Scudder, 1990] gives a general overview of E/T scheduling.

3.1.2 Splitting Jobs on Parallel Machines

Almost all scheduling research assumes that each job can be processed on at most one machine at a time, although preemption is allowed in certain cases. However, when the processing requirement is measured as a total demand of some product, the job fulfilling that demand can be arbitrarily split over a number of parallel machines and processed simultaneously to complete the job at the optimal time. Potts and Van Wassenhove [Potts and Van Wassenhove, 1992] referred to this process as lotstreaming or lotsizing, and the split parts as continuous sublots. Serafini [Serafini, 1996] applied this model to a scheduling problem in the manufacture of fabric in the textile industry.

Xing and Zhang [Xing and Zhang, 2000] used the term "splitting" instead of "lotsizing". They looked at the problem of splitting jobs on identical parallel machines ($\alpha \in \{P, Q\}$) with and without setup times, considering various optimality criteria γ. They proved that the problem of splitting jobs on identical parallel machines ($\alpha \in \{P, Q\}$) and $\gamma \in \{C_{max}, \max L_j, \sum C_j, \sum U_j, \sum w_j C_j\}$ without setup times and early penalties is polynomially solvable. However, where $\gamma \in \{\sum T_j, \sum w_j U_j\}$, the problem is NP-hard. The case when ($\alpha = P$, $\gamma = C_{max}$) with setup times (but without early penalties) is NP-hard [Xing and Zhang, 1998]. For this problem, they proposed a heuristic which has a worst-case performance ratio $\leq (7/4 - 1/m)$ ($m \geq 2$), by using the maximum completion time estimation procedure and setup time list scheduling.

3.2 The Early-Tardy Distinct Due Date Machine Scheduling Problem with Splitting Jobs and Setup Times (ETDDDsplit)

In real life, a problem similar to both the problems described above (the ET scheduling problem and the scheduling problem with splitting jobs) occurs in the manufacturing industry and in logistics. Often, jobs can be split into arbitrarily small subsections. Each subsection can then be processed simultaneously on separate processors, provided some cost (“setup time”) is paid for each such subsection. This scheduling problem can be described as splitting jobs on parallel machines where $\alpha \in \{P, Q\}$. The optimality criteria γ needs to consider both early and late penalties in completing a job before or after its due date given by the entity placing the job order. The early penalties represent the manufacturer’s cost of storing the finished goods before handing off the goods at the given due date. The late penalties represent the cost of the failing to meet the set deadline, which ranges from an actual imposition of monetary cost, to the loss of customer goodwill and reputation within in the industry.

This problem was based on a real-life scenario, where a T-shirt manufacturer receives orders to produce T-shirts. Each order (which corresponds to a job J_j) must have the following parameters :

- The number of T-shirts required. This corresponds to the processing requirement p_j of the job.

- A few templates (≥ 1) of the design the machines are to copy onto the T-shirts. The number of templates given to the manufacturer limits the number of subsections of the job that be run simultaneously, as each machine processing a subsection needs a template to work from. This parameter is described as $maxPara_{j.}$
- A due date or deadline when the finished T-shirts are required, d_j.
- A release time when the job is first made available for processing, a_i.
- A sequence-and machine-independent setup time, s_j. The processing machine cannot have any assigned job for s_j time units before it can start processing a subsection of J_j. This time is needed for the installation and configuration of the template and raw materials into the machine before processing can start. The machine-idle setup period must occur strictly after the completion of any other section that was processed earlier on the same machine.
- A form of early penalty for completion of the job before its due date. Ideally, the early penalty should reflect the cost of storing the finished goods prior to delivery. Thus, it should actually be proportional to the quantity of finished goods being stored after each section. For simplicity however, we assume a constant early penalty e_j which only has an effect if the job's completion time C_j is after the due date d_j.
  - Hence, the total early penalty for $J_j = \begin{cases} e_j \times (d_j - C_j) & for \quad (C_j < d_j) \\ 0 & otherwise \end{cases}$
- A late penalty for completing the job after the due date, t_j.
  - The total late penalty for $J_j = \begin{cases} t_j \times (C_j - d_j) & for \quad (C_j > d_j) \\ 0 & otherwise \end{cases}$

- Each job may have a limit to the number of subsections it can be split into. This parameter is denoted by $maxSplit_j$ (≥ 1).

The manufacturer has the following resources and knowledge :

- m parallel uniform machines M_i, each with its own machine rate r_i, $i \in \{1, \ldots, m\}$. Any machine can work on any job, albeit at different processing rates. The setup time required for each job is independent of the machine. Each machine can only process 1 subsection of any job at a time.
- The number of people (termed machine "servers") who do the setups for each job subsection. Each "server" can only set up 1 machine at a time, can do the setup for any job on any machine, and is kept occupied for the entire setup period. This limits the number of job subsections that can have their setups overlap or run concurrently. We describe this parameter as $maxServ$.
- A maintenance schedule for the machines. Each maintenance "job" J_{mj} includes
    - a specific machine M_i to which it must be assigned
    - a fixed time length p_{mj} which denotes how long it takes to conduct maintenance work on the machine. This value is independent of the machine rate. It is similar to the processing requirement of standard jobs, as if all machine rates $r_i = 1$.
    - a narrow time window during which the maintenance job must be scheduled. In this case, it was always fixed at $d_{mj} - a_{mj} = 1.5 * p_{mj}$ and t_{mj} should be infinitely (or for practical purposes, just very) large.
    - There is no need for early penalties for maintenance jobs, as it does not matter if it finishes before its deadline. Hence, $e_{mj} = 0$.

    - For all maintenance jobs, $maxSplit_{mj} = 1$, $maxPara_{mj} = 1$ and $s_{mj} = 0$.
    - Maintenance jobs on the same machine cannot overlap.
- There can also be a breakdown schedule for the machines. In real life, breakdowns obviously do not make themselves known in advance to schedulers. It is added here merely to ensure that not all machines are always available. Each breakdown "job" J_{bj} has
    - a specific machine M_i to which it must be assigned
    - a fixed time length p_{bj} which denotes how long the machine is broken down. In effect, this value is similar to a maintenance job's p_{mj}.
    - a time window during which the breakdown job must be "scheduled". Hence, $d_{bj} - a_{bj} = p_{bj}$ and e_j and t_j are infinitely large.
    - For all breakdown jobs, $maxSplit_{bj} = 1$, $maxPara_{bj} = 1$ and $s_{bj} = 0$.
    - Breakdowns and maintenance jobs on the same machine cannot overlap.

In summary, the scheduling problem with splitting jobs on uniform parallel machines and early and tardy penalties and distinct due dates (ETDDDsplit) can be defined by the following characteristics :

global variables

- n jobs J_j, $j \in \{1, \ldots, n\}$
- m machines M_i, $i \in \{1, \ldots, m\}$
- $maxServ$
- A set of maintenance jobs J_{mj} and breakdown jobs J_{bj} (further defined below)

job variables (for non-maintenance and non-breakdown jobs)

- processing requirement p_j
- fixed setup time s_j
- release time a_j
- due date d_j
- completion time C_j
- early penalty e_j
- late penalty t_j
- maximum number of subsections, $maxSplit_j$
- maximum number concurrent subsections, $maxPara_j$

job variables (maintenance or breakdown jobs)

- fixed time requirement p_{mj} or p_{bj}
- release time a_{mj} or a_{bj}
- due date d_{mj} or d_{bj}
- machine M_i which the job must be scheduled to run on

machine variables

- machine processing rate r_i

optimality criteria

- $\min\left(\sum_{j=1}^{n} penalty\ \ for\ \ job\ J_j\right)$ where ($penalty$ for J_j) = $\begin{cases} e_j\left(d_j - C_j\right) if\ \left(C_j < d_j\right) \\ t_j\left(C_j - d_j\right) if\ \left(C_j > d_j\right) \\ 0\quad if\quad \left(C_j = d_j\right) \end{cases}$

Chapter 4

A Review of Basic Scheduling Heuristics

There have a variety of heuristics which have been catered to solve scheduling problems. When there are no early penalties, a possible starting point is to schedule the jobs as early as possible every time a machine is free. With both early and late penalties present, there is a need to try and complete the most heavily weighted jobs as close to the deadline as possible to minimize the penalties. Some techniques that have been applied to forms of scheduling closely related to ETDDDsplit scheduling are briefly described below.

4.1 Greedy Techniques

The obvious approach to solving a scheduling problem (assigning some job to a machine once it becomes available) can be refined by giving each job some priority based on the optimality criteria. The job with the highest priority then becomes the next one to be assigned provided no constraints are violated. The simplest prioritizing dispatch rules include

- Shortest Processing Time (SPT) which schedules the jobs by non-decreasing processing requirement; there is also Longest Processing Time (LPT) which schedules the jobs in the opposite order to SPT
- Shortest Remaining Processing Time (SRPT) which makes allowances for preemption by checking at each point in time for the job with the smallest remaining processing requirement
- Earliest Due Date (EDD) which schedules the jobs by non-decreasing due date

Problems that allow preemption of jobs need some heuristics to break up the jobs into subsections. To that end, there have been some splitting and scheduling heuristics with performance assurances) such as

- Largest-setup-time list scheduling and splitting heuristic (LSU) [Monma and Potts, 1993]
- Largest-batch-time list scheduling and splitting heuristic (LBT) [Chen, 1993]
- Maximum completion time estimation procedure and set-up time list scheduling (ML) [Xing and Zhang, 1998]

There has been research on heuristics (prioritizing methods) for early-tardy scheduling problem with distinct due dates (ET DDD) that have no inserted idle time, such as Valente and Alves' [Valente and Alves, 2003] WPT-MS heuristic (weighted processing time and minimum slack). However, the optimal solution for the ET DDD problem requires that idle time be inserted between jobs [Baker and Scudder, 1990]. Moreover, in the case of the ETDDDsplit problem, the calculations determining job priority should consider the early and late penalties, the processing size remaining and the amount of idle machine time left between the job release time and the job deadline (the job's slack). The ETDDDsplit problem is also preemptive, so the priority rating of a job can change as it is being processed, causing the preemption of the currently running job for another of now-higher priority. That the ETDDDsplit problem has more than 1 machine also complicates matters.

As optimal solutions for ET DDD scheduling problems require idle time to be inserted between jobs, Baker and Scudder [Baker and Scudder, 1990] suggested that the problem be broken down into 2 distinct phases :

- A sequencing phase which orders the jobs
- A scheduling phase which completes the schedule by inserting idle time

4.2 Enumerative Techniques

Enumerative methods such as integer programming, mixed-integer programming, dynamic programming and branch-and-bound have been applied to scheduling problems. Generally, they have had limited success on NP-complete scheduling problems of large size, although Lagrangian relaxation and decomposition strategies help to increase the size of problems that can be handled within a reasonable time using mathematical techniques. Chen and Powell [Chen and Powell, 1995] used a mixture of decomposition methods, integer programming and branch-and-bound on parallel machine scheduling problems (with tardy penalties but not early penalties) of up to 100 jobs on 10 machines.

Hoogeveen and van de Velde [Hoogeveen and van de Velde, 1996] applied branch-and-bound to the single-machine ET DDD scheduling problem, and concluded that direct application to the problem posed problems. Firstly, the insertion of idle machine time is a valid means to reduce the early penalties, but its inclusion in the algorithm complicates the design substantially. Secondly, it is difficult to compute strong lower bounds for the algorithm, which greatly reduces its effectiveness.

4.3 Neighborhood Search Techniques

In comparison to enumerative techniques, general neighborhood search techniques can often be used to generate a viable, though probably sub-optimal schedule. They can also be easily enhanced when combined with other heuristics. Among the most popular of these are Tabu search, simulated annealing and genetic algorithms.

4.3.1 Tabu Search (TS)

Tabu search was developed by Glover [Glover, 1989; Glover, 1990] as a search technique for solving a wide variety of NP-hard problems. The basic idea is to explore the local search space of feasible scheduling solutions by a sequence of moves. Like gradient based techniques, a move from 1 schedule to another is made by evaluating all candidates and choosing the best available, even if the move temporarily results in a slightly inferior solution. Some moves are classified as tabu as they could lead to cycling, or is known to trap the search at a local minimum. These moves are placed on a Tabu List for a certain period of time (a short-term memory function), until the current solution has moved sufficiently far away from its original position. In large problems, Reeves [Reeves, 1993] suggested a candidate list strategy to identify and evaluate a subset of neighbors which may contain better solutions. This allows the search to move to new solutions more quickly, but at the cost of quality.

Three common neighborhood schemes for TS as applied to scheduling problems are :

- Adjacent pair-wise exchange (a job may be swapped with jobs directly before or after it in the schedule)
- Swap (any 2 jobs in the schedule may be swapped)
- Insert (a job is taken from its current position and placed in a another position in the schedule)

These can easily be used for the sequencing phase of the ET DDD problem. TS is usually applied on some starting job sequence based on greedy criteria, then idle time optimally inserted. There have been several idle time insertion procedures created for the scheduling phase [Fry et al, 1987; Yano and Kim, 1991].

James and Buchanan [James and Buchanan, 1997] combined the sequencing and scheduling phases by not overtly inserting idle machine into a job sequence, but rather by designating jobs as either early or tardy. They then used known characteristics of the common due date problem by Baker and Scudder [Baker and Scudder, 1990] (the optimum schedule is V-shaped with no idle time between jobs) as guidelines to schedule all early jobs in ascending e_j/p_j order, then all tardy jobs in t_j/p_j order. A neighbor in TS was thus defined to be a schedule that had one of its jobs' status changed from early to tardy or vice versa.

4.3.1a Long-term memory in Tabu Search

Long term memory in TS records attributes of the best solutions for

- diversification, to move the search to another area of the solution space, then
- intensification, to intensify the search in areas which have share some attributes with the best solutions so far.

James [James, 1998] explored the effectiveness of long-term memory in TS. Three types of data were stored in long term memory in this investigation – the number of times a jobs was positioned in a given location, the number of times a job was designated as early or tardy, and storing the entire search and all the solutions visited in a special memory structure. He concluded that long term memory does have a significant impact on search performance although more information stored does not guarantee better results. It was also noted that best results came from restarting the search from different starting points, widening the search space.

4.3.2 Simulated Annealing (SA)

Simulated annealing [Kirkpatrick et al, 1983] was derived from the physical process of annealing a liquid to its solid state. At its initial high temperature, the molecules in a liquid are disorganized and of random orientation. Liquids that are cooled quickly solidify with its molecules staying in their random orientation, resulting in a relatively high energy state. This is analogous to a problem solution being caught in a local minimum. Liquids that are cooled very slowly allow the molecules to shift into more ordered configurations, maintaining a state of thermodynamic equilibrium. The solid formed would thus reach the lowest energy state (and the problem solution that is found using infinitesimally small improvements would likewise be at the global minimum). The objective function for the problem solution is used as its "energy value". The probability a neighboring candidate solution is chosen as the basis for the next iteration is proportional to $\exp\left(-\left(\Delta E_{best} - \Delta E_{candidate}\right)/TK\right)$ where

T = the current temperature (which drops slowly with each iteration)

ΔE_{best} = the greatest possible improvement of the objective function

$\Delta E_{candidate}$ = the improvement for the neighboring candidate solution

K = normalization factor

Since the best neighbor is not always chosen, the algorithm is able to escape local minima. Rabadi et al [Rabadi et al, 2002] used SA to generate solutions to small- and medium-sized instances (up to 25 jobs) of the single-machine ET CDD problem with sequence-dependent setup times and no preemption. The results were very satisfactory compared to actual optimal solutions which had been found using branch-and-bound.

4.3.3 Genetic algorithms (GA)

Genetic algorithms were developed by Holland [Holland, 1992] as artificial adaptive systems to simulate natural evolution and mutation. Each solution corresponds to an individual of a species, and an evaluation function determines which ones survive to the next generation. It has been found to be a practical approach to generating good solutions to difficult problems. The basic mechanics of a GA are conceptually simple, though it may be difficult to state the problem and solution space in terms that GA can be applied onto. After creating an initial population of valid solutions, the algorithm :

- Maintain a population of valid solutions coded as artificial chromosomes of fixed length that have been rated as the “fittest” individuals
- Select 2 of the better solutions for recombination (“crossover”) to generate valid “offspring” solutions. Hopefully, the new solutions will carry good traits from both parents
- Perform mutation and other variation operators on the offspring
- Use these offspring to replace the poorer solutions in the population to improve it overall fitness, or create an new population altogether for the offspring.

Sivrikaya-Serifoglu and Ulusoy [Sivrikaya-Serifoglu and Ulusoy, 1999] used GA to tackle the ET DDD uniform parallel machine scheduling problem and sequence-dependent setup times with encouraging results. Each population member was described by a set of *n* genes, each gene containing an object-selection pair and encoding the associated job and the machine selected for its processing. The order of the genes on the chromosome dictated the order of the jobs on the machines, i.e. for n = 4, m =2, chromosome [3-2, 2-1, 4-1, 1-2] denoted that machine 2 processes job 3 then job 1, and machine 1 processes job 2 then job 4. The offspring chromosome was not constructed from single “crossover” operation, but by

- choosing one of the parents randomly
- finding the earliest gene/object on the parent that has not yet been assigned to the child and setting that gene as the next one on the chromosome
- picking from either of the parents the machine selection for the child

  iteratively until the child’s chromosome is complete. The mutation operators were :

  - the swap mutation, where 2 jobs’ positions are swapped
  - the bit mutation, where a job’s assigned machine is reassigned to a random machine (possibly the original machine)

They concluded that GA worked well for difficult scheduling problems, needing relatively modest processing requirements and that it could be scaled effectively for larger problems.

4.4 Artificial Intelligence (AI) Techniques

Artificial intelligence uses knowledge of the problem to guide search for a solution, unlike the neighborhood searches' method of mostly blind search using an evaluation function.

4.4.1 Neural Networks (NN)

Neural networks attempt to simulate the learning and prediction abilities of the human brain. They are distinguished by network topology, node characteristics, and training rules for the network weights. Through training with supplied data, supervised learning neural networks attempt to capture the desired relationship between the inputs and the outputs of the network. Neural networks have been used more commonly for job-shop [Jain and Meeran, 1998] and flow-shop scheduling rather than ET machine scheduling.

Hamad et al.[Hamad et al, 2002] was first to use a multi-layer perceptron model to tackle the ET DDD problem on parallel identical machines. The network had 11 inputs for the 11-element vector used to represent each job, *m* outputs and a 9-node hidden layer. The output node with the highest value then designated the machine the job should be processed on. After all the jobs had been fed to the NN, the jobs were scheduled on their assigned machines in order of increasing output values. The network was trained using error back-propagation. The trained neural network gave near-optimal solutions, but it was run on very small cases of up to 6 jobs on 2 machines.

4.4.2 Multi-agent Systems (MAS)

An agent is a computer system that is capable of autonomous decisions and social activity in some environment to meet its design objectives. Gozzi et al.[Gozzi et al, 2002] created a MAS for the ET DDD scheduling problem on identical parallel machines. The MAS had job agents, machine agents, a contract coordinator agent, and agents to produce the job and machine agents. The job and machine agents all had their own agendas, and when created, a job agent was given a budget according to the job's urgency or weight. The job agents tried to minimize the ET penalties on their given budgets by bidding for machine time, and the machine agents tried to maximize their profit from the jobs, and the number of jobs processed. The coordinator agent managed the evolution of time and updates the partial schedule. The authors explored different agent behaviors and their effect on the final solutions created.

Chapter 5

Solving the ETDDDsplit problem

5.1 How hard is the ETDDDsplit problem?

The parallel machine scheduling problem where ($\alpha \in \{P, Q\}$, $\gamma = \sum T_j$) with splitting jobs and setup times is NP-complete [Xing and Zhang, 1998]. It can be shown that the ETDDDsplit scheduling is NP-complete too.

The ETDDDsplit scheduling problem is obviously in NP. The time taken to calculate the sum of the penalties for all the jobs given any valid schedule would be of $O(n)$ (involving 1 simple mathematical operation each job's C_j, d_j, e_j and a_j values). The problem can be phrased as a decision problem by checking whether there exists a schedule for the given problem which has a total penalty that is less than some value.

By restriction [Garey and Johnson, 1979], the ETDDDsplit scheduling problem can be shown to be NP-complete, as it can be transformed into the above known NP-complete problem by applying the following restrictions :

- the tardy penalty is much larger than the early penalty for all jobs ($t_j >> e_j$)
- the release dates of the jobs are all zero ($a_j = 0$)
- the due dates of all the jobs are the same ($d_{j1} = d_{j2}$ for all $j1, j2$ where $1 \leq j1 < j2 \leq j$)
- the set of maintenance jobs is empty
- the set of breakdown jobs is empty

5.2 Heuristics for the ETDDDsplit scheduling problem

The ETDDDsplit problem is a completely new problem that carries traits from 2 mostly disparate scheduling problems currently existing in literature - the ET scheduling problem, and the scheduling problem with splitting jobs. Not all the approaches described Chapter 4 were applied to the ETDDsplit problem. The ones applied were a greedy heuristic, tabu search, simulated annealing and genetic algorithm. As described earlier, many approaches to dealing with ET DDD scheduling problems were divided into 2 phases; the first phase involved some form of job ordering, and the second phase was where the solution schedule was actually created using that job ordering, with the insertion of idle machine times that are necessary for any (near) optimal solution. The heuristic described in this chapter follow this basic idea.

5.2.1 Greedy Scheduling Heuristic

The first and obvious approach to the problem was to apply a some “greedy” scheduling heuristic to the jobs, This can be done in 2 phases : by first assigning a priority rating to each job which would dictate the overall order of importance in which the jobs would be scheduled, and then finding a way to create a schedule using the job ordering. The priority ratings would change as jobs are processed. Hence, other waiting jobs’ priority ratings might overtake the current jobs’ ratings, causing them to be preempted. The scheduling heuristic utilizing priority ratings (R_j) should consider :

- e_j and t_j (both the penalties). For this problem, e_j was in general set much lower than t_j but still non-zero. This reflected the general preference to finishing a job early rather than late.

- the remaining unassigned processing requirement of the job (rp_j)
- the slack, or the amount of processing power still available due to unassigned machine time between the job's release time and deadline ($slack_j$). This was actually the unassigned machine time multiplied by the respective machine rates r_i except for maintenance jobs, which treated all machine rates $r_i = 1$.
- the number of subsections ($sections_j$) already created, compared to $maxSplit_j$. If there was little leeway left to split rp_j into subsections, the job was considered more urgent.

Some of these characteristics could be related to each other. The priority rating eventually used was :

$$R_j = k_1 t_j \left(\frac{rp_j + s_j}{slack_j + 1} \right) + k_2 e_j \left(\frac{\max Split_j}{1 + \max Split_j - sections_j} \right)$$

where

k_1, k_2 = constants used to adjust the relative importance of the 1st and 2nd terms.

In the 1st term, t_j was associated with rp_j and $slack_j$, as the extent to which a job would likely be tardy could be directly linked to the amount of time it had available to be completed. Hence, the tardy penalty would be considered more important if the slack is small. In the 2nd term, e_j was associated with the freedom still available to the scheduler to schedule the job. If the scheduler still had a great deal of freedom to break up the job into subsections, it would be more likely to have the job be completed close to the due date. However, if the scheduler was strongly limited by $maxSplit_j$, the job could be forced to complete in a single large block well before its due date. For both the 1st and 2nd terms' denominators, some care had to be taken as $slack_j$ and ($maxSplit_j$ – $sections_j$) could easily equal zero. Hence, the value of 1 was added to both denominators to prevent problems in calculations and comparison between each job's R_j value.

A higher rating R_j means that job J_j is currently more urgent. A job which has been completed automatically has $R_j = 0$. As $t_j >> e_j$, the 1st part of the equation would generally have a much greater effect on R_j than the 2nd unless $sections_j \approx maxSplit_j$. A larger $slack_j$ or smaller rp_j reduced the overall priority of the job.

Since there are both early and tardy penalties, instead of working "forwards" in time by placing jobs in a sequence and then inserting idle time to reduce early penalties, an alternative method was to work "backwards" in time, with the starting point of scheduling a job being its deadline. The sequencing and splitting of jobs could then be combined by using the priority ratings to select jobs to schedule. A job would be automatically split if another job's priority rating overtook it at some point in time. Idle time would be automatically inserted to allow a job to finish at its deadline rather than before. This method would obviously work only if most or all the jobs were known long before their respective due dates.

In the course of scheduling the current (and most urgent) job J_{urgent}, there would be periodic comparisons of J_{urgent}'s priority rating with that of other jobs'. After scheduling J_{urgent} to be processed on any machine for a certain time interval (termed "checktime"), the current job's priority rating was checked with that other jobs. During a "checktime", a job-in-progress could be preempted for another job with a higher priority rating. The pseudo code for the greedy scheduling heuristic is in Fig. 1 and 2.

```
1. Place all "breakdown" jobs $J_{bj}$ in the schedule. All
   breakdown jobs are tagged as "complete", all non-
   breakdown jobs are tagged as not "complete".

2. While (not all jobs complete)
      a. Update all $R_j$ values. Find incomplete job $J_{urgent}$ with
         highest $R_j$.
      b. Set $best\_section$ = null.
      c. If there exists idle machine time between $d_{urgent}$ and
         $a_{urgent}$
          i. While-loop through the machines from machine with
             highest to lowest machine rate $r_i$. Note that if
             $J_{urgent}$ is a maintenance job, the machine to be used
             is fixed.
             A. Initialise $t_{end}$ to the latest idle time on the
                current machine before $d_{urgent}$ but after $a_{urgent}$
                and set $section\_found$ = false
             B. While $section\_found$ = false and
                $t_{end}$ > ($a_{urgent}$ + $s_{urgent}$)
                 I.  Find the longest continuous section on the
                     current machine that starts at $t_{earliest}$ and
                     ends at $t_{end}$ within which no violation of
                     $maxPara_{urgent}$ would occur.
                II.  If the section length < $s_j$, set $t_{end}$ =
                     $t_{earliest}$ then continue search.
               III.  If
                     • $sections_{urgent}$ = $maxPara_{urgent}$ - 1 or
                     • $J_{urgent}$'s total remaining processing time on
                       the current machine ≤ checktime + $s_{urgent}$ or
                     • $J_{urgent}$'s total remaining processing time on
                       the current machine ≤ 2 * checktime
                       but $J_{urgent}$ cannot be completed in this
                       section, set $t_{end}$ = $t_{earliest}$ then continue
                       search.
                IV.  Else (with respect to II and III)
                     Attempt to schedule a subsection of $J_{urgent}$
                     (See Fig. 2 for details).
                     •  If a job subsection can be scheduled,
                        set $section\_found$ = true and update
                        $best\_section$ if the newly found
                        jobsection's endtime is later than
                        $best\_section$'s end time.
                     •  Otherwise set $t_{end}$ = $t_{earliest}$ then
                        continue search.

      d. If $best\_section$ = null, schedule this job to
         complete as quickly as possible after $d_{urgent}$ without
         further splitting.
      e. Else, insert $best\_section$ into the schedule.

      f. Update the characteristics of all jobs.
```

Fig. 1 Pseudo code for the greedy scheduling heuristic (overall)

```
1. Set $t_{start}$ = -1 and $t_{opt\_start}$ = $t_{end}$ - ($s_{urgent}$ + $rp_{urgent}/r_i$).
2. If
      • $sections_{urgent}$ = $maxSplit_{urgent}$ - 1 or
      • $J_{urgent}$'s total remaining processing time on the
        current machine ≤ checktime + $s_{urgent}$ or
      • $J_{urgent}$'s total remaining processing time on the
        current machine ≤ 2 * checktime
   set $t_{start}$ = $t_{opt\_start}$.
      a. If $maxServer$ is violated, set $t_{start}$ earlier but
         always later than $t_{earliest}$ so that no violation
         occurs. If this is not possible, set $t_{start}$ = -1.
3. Else set $t_{start\_test}$ = $t_{end}$ - checktime, $searching$ = true.
      a. While $searching$ = true and $t_{start\_test}$ ≥ $t_{earliest}$
           i. Set $valid\_try$ = true.
          ii. If $maxServer$ is violated, set $t_{start\_test}$
              earlier so that no violation occurs. If this
              is not possible, try later start times. If
              neither works, set $valid\_try$ = false.
         iii. If $valid\_try$ = true, set $t_{start}$ = $t_{start\_test}$.
                I. Calculate temporary values for $rp_{urgent}$ and
                   the slack for all other incomplete jobs
                   assuming the current section is
                   processed. Calculate temporary values of
                   all the incomplete jobs' $R_j$ values and
                   compare the temporary $R_{urgent}$ value with
                   them.
               II. If the temporary $R_{urgent}$ is not one of $m$
                   highest priority ratings,
                   set $searching$ = false
          iv. If $t_{start\_test}$ - $t_{opt\_start}$ ≤ (checktime or $s_j$),
              set $t_{start\_test}$ = $t_{opt\_start}$ and $searching$ = true.
           v. If $t_{start\_test}$ - $t_{earliest}$ ≤ (checktime or $s_j$),  set
              $t_{start\_test}$ = $t_{earliest}$ and $searching$ = true.
          vi. Else  (to iv and v)
              decrease $t_{start\_test}$ by checktime units.

4. If $t_{start}$ > -1 and $t_{start}$ < $t_{opt\_start}$
   set $t_{end}$ = $t_{start}$ + $s_{urgent}$ + $rp_{urgent}/r_i$.

5. Return to Fig. 1 indicating success ($t_{start}$ > -1) with
   details of the section found, or failure ($t_{start}$ = -1).
```

Fig. 2 Pseudo code for the greedy scheduling heuristic (job subsection placement)

Some comments on the overall greedy scheduling heuristic, for Fig. 1 :

- (Step 1) The breakdown jobs were first inserted into schedule, blocking out the unavailable slots from the start. Al non-breakdown jobs were flagged as incomplete.

- (Step 2a) In each iteration of the Step 2 while-loop, one job subsection of a currently incomplete job was definitely scheduled in Step 2d or 2e. This had an effect on one job's rp_j, $sections_{urgent}$ and possibly its "complete" status. Simultaneously, all jobs' $slack_j$ could be affected. These changes were updated in Step 2f. Hence, all jobs' R_j values were updated at this step before the next selection of J_{urgent}.
- (Step 2b) *best_section* contained the details of the best subsection found for J_{urgent} in the current iteration of the Step 2 while-loop, including which job being processed, the machine involved, and the start and end times of the subsection.. The best subsection was considered to be the subsection with the latest end time on the fastest available machine.
- (Step 2c & 2c(i)) Tried to schedule at least a subsection of J_{urgent} before d_{urgent}. The fastest machine was checked first before moving on to the next fastest machine until all machines had been considered. Preferentially using the machine with the highest machine rate for the most urgent job tended to ensure that the machines with high machine rates would be used more.
- (Step 2c(i)-A & B) A search was made on the current machine based on t_{end}. *section_found* indicated whether a suitable job subsection had been found for the current machine. The search backwards in time continues until a suitable section had been found, or until t_{end} reached too small a value to be useful.
- (Step 2c(i)-B I) The longest continuous section found always ended at the current t_{end}, and could start at any time between t_{end} and a_{urgent} (any time before J_{urgent} had been released could not be considered).
- (Step 2c(i)-B II & III) If the length of time that a machine was idle was too short, t_{end} was decremented and the search continued without further checks,

skipping Step 2c(i)-B IV. If the section length was less than the setup time, no subsection of J_{urgent} could even be processed. If J_{urgent} could or should not be split into any more subsections, the length of time that a machine was idle had to be sufficient for J_{urgent} to be completed ($\geq s_j + rp_j/r_j$) in order to schedule the job on that machine at that time. J_{urgent} should not be further split if the processing time was less than 2*checktime or s_j+checktime. Note that a job J_j's processing time on M_i is not equal to rp_j (remaining processing requirement). It is actually rp_j/r_i, the amount of machine M_i's time that is required to complete the job. The heuristic would try to minimize unnecessary job splitting, as each split results in "wasted" machine time due to the additional setup period incurred for each subsection. The most efficient use of machine time would be for each job to have only 1 subsection and hence only 1 setup period.

- (Step 2c(i)-B IV) At this point, a potential place in the schedule had been found to place at least a subsection of J_{urgent}. Further checks were required to determine the exact length and position of the section, and are described in Fig. 2. If the pseudo code in Fig. 2 could successfully place a job subsection in the schedule, the current iteration of the Step 2c(i)-B while-loop would be essentially completed, i.e. a suitable section of J_{urgent} had been found for the current machine. This does not necessarily mean that J_{urgent} could be completed with the newly found section, only that at least a subsection of J_{urgent} could be inserted into the schedule before d_{urgent} on the current machine. The found section would be compared with the best section found so far on another machine (recorded by *best_section*), and the section with the later end time stored in *best_section*.

- (Step 2d) If *best_section* was empty, this meant that no part of J_{urgent} could be scheduled before d_{urgent} in the current iteration of Step 2. J_{urgent} would have to be completed after d_{urgent} and forced to be completed as soon as possible. Note that it is still possible for some subsection of J_{urgent} to have been placed in the schedule before d_{urgent} earlier on in the overall scheduling process.
- (Step 2f) As stated in the comments for Step 2a, the characteristics of all jobs (not just of J_{urgent}) would have to recalculated after the insertion of a job subsection into the schedule. J_{urgent} would be flagged as complete only if the all the job subsections that had been inserted into the schedule so far (including the one just inserted in this latest iteration) was sufficient to finish processing the job. Once flagged as complete, it would no longer be considered for selection as J_{urgent} in future iterations of Step 2a.

Further comments on the greedy scheduling heuristic involving job subsection placement, for Fig. 2 :

- This pseudo code checks the current machine for a suitable place to insert a subsection of J_{urgent}, using t_{end} as a starting point and both t_{end} and $t_{earliest}$ as restrictions.
- (Step 1) t_{start} contains to the best <u>valid</u> start time found so far for the subsection being placed. At the end of the pseudo code in Fig. 2, as long as t_{start} was a value other than “-1”, a subsection of J_{urgent} could be successfully scheduled. t_{opt_start} referred to the start time that would allow J_{urgent} to be completed (given t_{end}) with this subsection, temporarily ignoring $t_{earliest}$ as a restriction.
- (Step 2) These are the same conditions as stated in Fig. 1 Step 2c(i)-B III.

- (Step 2a) Since *maxServer* only affects the setup period of any subsection, moving the start time of the subsection tried to fix this violation. The value of t_{start} must always be later than $t_{earliest}$. t_{start} could not be moved later than its original value as it would then be impossible for J_{urgent} to be completed with this subsection.
- (Step 3 and 3a) A more involved set of checks were required if J_{urgent} could be further split. t_{start_test} was used as an interim value in checking whether J_{urgent} should be split after partial processing. The "checktime" value was the size of the steps used in a step-wise comparison with other jobs. *searching* was used as the flag to stop or continue the while-loop search for t_{start}. The search for t_{start} was limited to the values between $t_{earliest}$ and t_{end}–checktime.
- (Step 3a(ii)) This is similar to Step 2a, except that now t_{start_test} could also be moved to a later time, though no later than $t_{end}-(s_{urgent}+1)$.
- (Step 3a(iii)) t_{start} was updated to store the best valid start time found so far the current J_{urgent} subsection.
- (Step 3a(iii)-I & II) The calculation of many temporary values was required to compare R_j values, much as if the section had already been scheduled using t_{start} and t_{end}, and ensuring $rp_{urgent} \geq 0$. The search for a better t_{start} was usually stopped if the effective value of R_{urgent} has dropped past the top m values (the exception to this is indicated in the next step). Since there can be up to m different jobs processed simultaneously, as long as R_{urgent} remained high enough compared to other R_j, there was less need to split the job and incur the additional setup costs.
- (Step 3a(iv), (v) & (vi)) If J_{urgent} was close enough to completion or there was very little available processing time left on the machine, then the search

should continue regardless of the results of Steps 3a(ii) & (iii). This was again to prevent additional setup costs. t_{start_test} was decreased by checktime, or to t_{opt_start} or $t_{earliest}$ depending on the circumstances. The conditions of the Step 3a while-loop would prevent the search if $t_{opt\_start} < t_{earliest}$.

- (Step 4) t_{end} was set to an appropriate value if the t_{start} adjustments made in Steps 2a or 3a(ii) to remove *maxServer* violations made the job subsection longer than would be required to complete J_{urgent}.
- (Step 5) If a subsection of J_{urgent} could be placed in the schedule on the current machine, t_{start} would have a non-"-1" value. The section start and end time values of the suitable section found would be returned to the pseudo code of Fig. 1.

A simple example running through 1 iteration to schedule a job subsection is described below. Consider a small scheduling problem : 3 machines M_A, M_B and M_C where $r_A > r_B > r_C$ and 4 jobs J_1, J_2, J_3 and J_4. At some point in the scheduling process, some of the machine time has been taken up by job subsections already scheduled, shown as shaded sections in the diagram below. A short section showing the time period of interest (involving J_{urgent}) of the 3 machines is shown in Fig. 3.

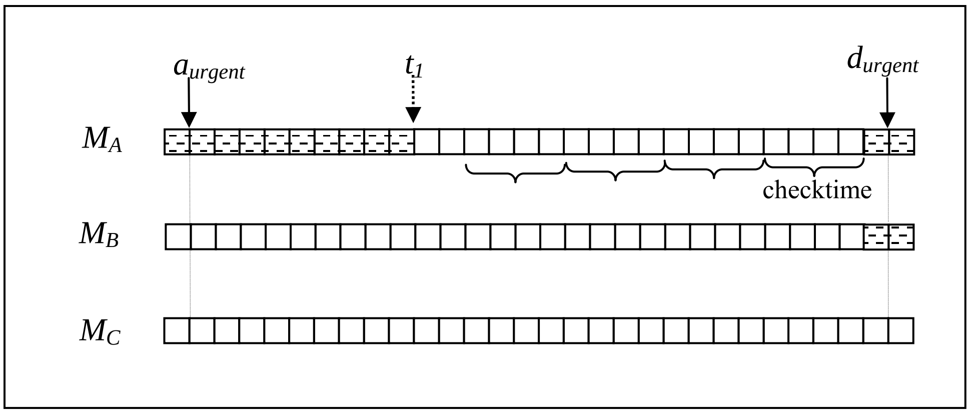

Fig. 3 Current machine schedule for greedy heuristic example

Assume that $J_{urgent} \equiv J_4$, and that $sections_{urgent} < maxSplit_{urgent} - 1$. In Fig. 3, M_A and M_B are only idle at $d_{urgent} - 1$, M_C is idle at d_{urgent}. Assume that there is sufficient processing time available on M_B to complete J_{urgent} at $d_{urgent} - 1$ and on M_C at d_{urgent}, but also that J_{urgent} cannot be completed on M_A in the time between t_1 and $d_{urgent} - 1$ although $d_{urgent} - 1 - t_1$ is much larger than s_{urgent} (allowing the idle time of M_A to be searched).

The pseudo code in Fig. 1 would first attempt to schedule a subsection of J_{urgent} on M_A with $t_{earliest} = t_1$ and $t_{end} = d_{urgent} - 1$. Fig. 2 is then called to exactly describe a section that could be scheduled on M_A. t_{opt_start} would be calculated to be earlier than $t_{earliest}$. The length of the subsection would first be set to checktime ($t_{start\_test} = d_{urgent} -$ checktime $- 1$), *maxServer* checked for any violation and then the priority ratings of all the jobs recalculated using new values of $slack_j$ and a new value of rp_{urgent} and $sections_{urgent}$. Assuming that the priority rating of J_4 was one of 3 highest values, the length of the subsection would be set to 2 * checktime and the priority ratings recalculated again. This could continue until subsection length = 4 * checktime. At the next iteration, since the remaining available processing time would be less than checktime, t_{start_test} would be set to t_1 and checked to make sure no *maxServer* violations occurred. A success would be returned to the Fig. 1 pseudo code, and *bestSection* updated to reflect a section on M_A starting at t_1 and ending at $d_{urgent} - 1$, even though J_{urgent} would not be completed with this new section.

The Fig. 1 pseudo code would proceed to the next fastest machine M_B, with $t_{earliest} = a_{urgent}$ and $t_{end} = d_{urgent} - 1$. In this case, Fig. 2 would return a success with the section on M_B starting at t_{opt_start} calculated in Fig. 2 and ending at $d_{urgent} - 1$. However, since the end time of this section was the same as that of *bestSection*, there would be no change in *bestSection*.

Finally, the slowest machine M_C would be searched, using $t_{end} = d_{urgent}$ and $t_{earliest} = a_{urgent}$. The pseudo code of Fig. 2 would be able to find a suitable section on M_C starting at t_{opt_start} and ending at d_{urgent}. Since the ending time of this section is later than that in *bestSection* (and not because the job can be completed with this new section), *bestSection* is updated.

With all the machines searched, the schedule would be updated with the job subsection details stored in *bestSection*, and all jobs' priority ratings updated with the new slack values, remaining processing requirements and number of job sections. The next J_{urgent} would have to be found (which could be J_4 again) in order to schedule the next job subsection.

The greedy scheduling heuristic could be randomized somewhat by multiplying some random modifier with each job's priority rating. The random modifier was changed each time a priority rating calculation was done and kept quite small. Hence, at any point during the scheduling, the modifier could allow a job J_{j1} whose rating R_{j1} was only slightly less than another job J_{j2}'s rating R_{j2} to be selected next for scheduling next in place of J_{j2}. This was the simplest version of searching the area of the solution space closest to the solution found by the greedy heuristic.

5.2.2 Adjusted Greedy Scheduling Heuristic

In order to implement broader and more directed searches from the initial greedy solution, the solutions first had to be represented in a manner that could be easily manipulated. We first observe that if the $maxSplit_j$ values are not very high, the initial priority rating tends to dictate quite strongly the eventual overall order that the jobs are scheduled. Although this assumption does not always hold, it serves as the basis for the adjustment described below. To apply the searches described later, a starting job queue was first derived from the initial priority rating of all jobs, with the head of the queue being the job with the highest R_j and subsequent jobs ordered by non-increasing R_j. The scheduling heuristic then needed to be changed to utilize this priority job queue. Jobs were removed from the queue upon completion. The pseudo codes for the adjusted greedy scheduling heuristic that used a priority job queue are shown in Figs. 4 and 5. They follow the format used to describe the greedy scheduling heuristic in section 5.2.1 with amendments to reflect the required changes. The main adjustment comes from only allowing a subset of the jobs (a "pool" of $m + 1$ jobs) to be available for scheduling at any time, instead of all jobs being available as in the original dispatch policy. With this heuristic, any initial queue of jobs could be used to generate a valid schedule.

```
1. Place all "breakdown" jobs $J_{bj}$ in the schedule. Create a
   queue of incomplete jobs in non-increasing order($q_{current}$);
   the job at the head of the queue has the highest $R_j$. All
   breakdown jobs are tagged as "complete", all non-
   breakdown jobs are tagged as not "complete".
2. Remove the first ($m$+1) jobs from the queue. These ($m$+1)
   jobs constitute the "pool" of available jobs from which
   $J_{urgent}$ can be selected.
3. While (not all jobs complete)
   a. Update the $R_j$ values of all jobs in the pool. Find
      incomplete job $J_{urgent}$ with highest $R_j$ of all jobs in
      the pool.
   b. Set best_section = null.
   c. If there exists idle time between $d_{urgent}$ and $a_{urgent}$
        i. While-loop through the machines from highest to
           lowest machine rate $r_i$.
           A. Initialise $t_{end}$ to the latest idle time on eth
              current machine before $d_{urgent}$ but after $a_{urgent}$
              and set section_found = false
           B. While section_found = false and
              $t_{end} > (a_{urgent} + s_{urgent})$
               I.  Find the longest continuous section on the
                   current machine that starts at $t_{earliest}$ and
                   ends at $t_{end}$ within which no violation of
                   $maxPara_{urgent}$ would occur.
              II.  If the section length < $s_j$, set $t_{end}$ =
                   $t_{earliest}$ then continue search.
             III.  If
                   • $sections_{urgent} = maxPara_{urgent} - 1$ or
                   • $J_{urgent}$'s total remaining processing time on
                     the current machine ≤ checktime + $s_{urgent}$ or
                   • $J_{urgent}$'s total remaining processing time on
                     the current machine ≤ 2 * checktime
                   but $J_{urgent}$ cannot be completed in this
                   section, set $t_{end}$ = $t_{earliest}$ then continue
                   search.
              IV.  Else (with respect to II and III)
                   Attempt to schedule a subsection of $J_{urgent}$
                   (See Fig. 5 for details).
                   •  If a job subsection can be scheduled,
                      set section_found = true and update
                      best_section if the newly found
                      jobsection's endtime is later than
                      best_section's end time.
   d. Otherwise set $t_{end}$ = $t_{earliest}$ then continue search.

   e. If best_section = null, schedule this job to complete
      as quickly as possible after $d_{urgent}$ without further
      splitting.
   f. Else insert best_section into the schedule.
   g. If $J_{urgent}$ was completed with the addition of the newest
      job subsection, remove $J_{urgent}$ from the available pool.
      If the queue is non-empty, remove the job at the head
      of the queue and place it in the pool.  Update the
      characteristics of all jobs.
```

Fig. 4 Pseudo code for the adjusted greedy scheduling heuristic (overall)

```
1. Set $t_{start}$ = -1 and $t_{opt\_start}$ = $t_{end}$ - ($s_{urgent}$ + $rp_{urgent}/r_i$).
2. If
      • $sections_{urgent}$ = $maxSplit_{urgent}$ - 1 or
      • $J_{urgent}$'s total remaining processing time on the
        current machine ≤ checktime + $s_{urgent}$ or
      • $J_{urgent}$'s total remaining processing time on the
        current machine ≤ 2 * checktime
   set $t_{start}$ = $t_{opt\_start}$.
      a. If $maxServer$ is violated, set $t_{start}$ earlier but
         always later than $t_{earliest}$ so that no violation
         occurs. If this is not possible, set $t_{start}$ = -1.
3. Else set $t_{start\_test}$ = $t_{end}$ - checktime, $searching$ = true.
      a. While $searching$ = true and $t_{start\_test}$ ≥ $t_{earliest}$
           i. Set $valid\_try$ = true.
          ii. If $maxServer$ is violated, set $t_{start\_test}$
              earlier so that no violation occurs. If this
              is not possible, try later start times. If
              neither works, set $valid\_try$ = false.
         iii. If $valid\_try$ = true, set $t_{start}$ = $t_{start\_test}$.
               I. Calculate temporary values for $rp_{urgent}$ and
                  the slack for all other incomplete jobs
                  in the available queue, assuming the
                  current section is processed. Calculate
                  temporary $R_j$ values of all the incomplete
                  jobs in the available pool and compare
                  the temporary $R_{urgent}$ value with them.
              II. If the temporary $R_{urgent}$ is the smallest of
                  the $R_j$ values checked,
                   set $searching$ = false
          iv. If $t_{start\_test}$ - $t_{opt\_start}$ ≤ (checktime or $s_j$),
              set $t_{start\_test}$ = $t_{opt\_start}$ and $searching$ = true.
           v. Else decrease $t_{start\_test}$ by checktime units.

4. If $t_{start}$ > -1 and $t_{start}$ < $t_{opt\_start}$
   set $t_{end}$ = $t_{start}$ + $s_{urgent}$ + $rp_{urgent}/r_i$.

5. Return to Fig. 4 indicating success ($t_{start}$ > -1) with
   details of the section found, or failure ($t_{start}$ = -1).
```

Fig. 5 Pseudo code for the adjusted greedy scheduling heuristic
(job subsection placement)

Figs. 4 and 5 for the adjusted greedy scheduling heuristic correspond almost exactly with those of the greedy scheduling heuristic described earlier (Figs. 1 and 2). The main differences and some comments are described below :

- (Fig. 4, Step 1) The priority job queue never included breakdown jobs, all of which were always scheduled before any other jobs in the heuristic. Once the order of the queue had been fixed at this step, it is not changed for the rest of the scheduling process. $q_{current}$ represents the job ordering used, and is subject to manipulation in the heuristics described later in this chapter.

- (Fig. 4, Step 2) A subset of all jobs, an available "pool" of jobs from which J_{urgent} is always selected, is used. This available pool always contains m+1 jobs until at least n–m jobs have been completed. If a job is completed after an iteration, it is removed from the pool, then job at the head of the queue is moved into the pool. In this way, the starting sequence of jobs in the queue created at Step 1 will have a very large impact on the actual order in which the jobs are eventually scheduled.
- (Fig. 4, Step 3) The whole section basically follows the same process the greedy scheduling heuristic earlier described in Fig. 1, Step 2. The main differences are that the selection of J_{urgent} is limited to the jobs in the pool (jobs still in the priority queue are not considered), and that updating the characteristics of the jobs after insertion of a new subsection includes the contents of the available pool and the queue.
- (Fig. 5, Step 3a(iii)-II) This step contains the only change from Fig. 2. At this point, the new (and temporary) value of R_{urgent} is compared to the new R_j values belonging to the jobs in the available pool only. The rest of the jobs in the priority queue are again not considered. Only if R_{urgent} is the smallest of all R_j values checked is J_{urgent} pre-empted for another job. As with the original greedy heuristic, the pre-emption is ignored if J_{urgent} is sufficiently close to completion.

A second way was also used to represent the relative urgencies of the jobs as given by the initial R_j. Instead of using a priority queue, an integer array of length n stored the order given by the initial R_j's. For example, if the j^{th} array integer value was 10, this meant that job J_j would have the 10th highest initial R_j value of all uncompleted jobs (i.e. non-breakdown jobs). All breakdown jobs' corresponding array values were set to -1 so that they would not be considered as part of the job queue. Like the queue, this array could also be acted upon by a search heuristic to give a valid schedule as long as all cells with "-1" values were left alone and no other integer contents were repeated. The use of an array instead of a priority queue was necessary for implementation of the genetic algorithm in section 5.2.5. The algorithm would be able to manipulate the starting point of the search - the order in which the jobs would be scheduled - by making changes to the array. The resulting manipulated array could then easily be transformed into its equivalent priority job queue, and the same adjusted dispatch policy used to generate a valid schedule.

The aim of creating the adjusted greedy heuristic was to create a template onto which different search techniques could be applied. The following sections of chapter 5 describe the application of tabu search, simulated annealing and genetic algorithm to the scheduling heuristic. In all these cases, the searches made changes in the initial ordering of jobs in the priority queue or its associated array. After the queue or array had been manipulated, the steps outlined in Figs. 3 and 4 would always be run on the manipulated queue ($q_{current}$ or its corresponding array) to generate the final schedule.

5.2.3 Tabu Search (TS)

Tabu search (without long-term memory) was used to broaden the search around the initial solution found by the greedy dispatch policy. The pseudo code for the TS heuristic used is shown in Fig. 6.

```
Set $queue_{current}$ = $queue_{from\ initial\ ratings}$;
    $queue_{best}$ = $queue_{current}$;
    $cost_{best}$ = cost of $queue_{best}$;

Repeat $n$ times
    bestNeighborFound = false;
        While (!bestNeighborFound)
            Create TS_NEIGHBOR neighboring solutions, each
            neighbor created by swapping 2 jobs in $queue_{current}$
            without violating the Tabu List;
            Calculate the penalties for all neighboring
            solutions;
            If (cost of best neighbor ≤ ALLOWANCE * $cost_{best}$)
                  Add best neighbor's swap to Tabu List;
                  $queue_{currnt}$ = best neighboring solution;
                  bestNeighborFound = true;
                  If (cost of $queue_{current}$ < $cost_{best}$)
                          $cost_{best}$ = cost of $queue_{current}$;
                          $queue_{best}$ = $queue_{current}$;

Return ($queue_{best}$) as the best solution;
```

Fig. 6 Pseudo code for the tabu search heuristic

Some comments on the TS heuristic :

- There are 3 variables that can be manipulated :
  - *TS_NEIGHBOR* is the size of a subset of the neighborhood of $queue_{current}$ that is searched. Unless no suitable solution can be found for the next step of the search from this subset, the rest of the neighboring space is not searched.
  - *TABU_TENURE* is the number of search steps a swap remains tabu.
  - *ALLOWANCE* is the leeway granted to the search heuristic in choosing a solution for the next step of the search. As long as a solution is at least

ALLOWANCE times within the best solution, it can be used as the focal point of the next search step.

- The job queue derived from the initial R_j values (as described earlier) was used as the starting point of the search, and initially set to be the best solution queue.
- To determine the cost of a given job queue, the adjusted greedy scheduling heuristic had to be run on the job queue. The sum of early and late penalties on the resulting schedule would be the cost of that job queue.
- For any point in the solution space, a neighbor was defined to be another job queue with 2 of the jobs' places in the original queue swapped. Hence, each solution job queue would have up to ${}^{n}C_{2}$ neighbors (slightly less, if there are breakdown jobs). To reduce search time, a random subset of size
- *TS_NEIGHBOR* of the full neighborhood was explored. If a suitable neighbor was found from this subset, it was chosen immediately without exploring more of the neighborhood. The best of this neighborhood subset would then be chosen as the basis of the next step. In practice, the maximum number of neighborhoods the heuristic searched was 100, in order to limit the processing time required.
- A suitable neighbor did not always have to improve on the best solution found so far. A slight worsening of the solution penalties with respect to the best solution found so far (by *ALLOWANCE* times) could be accepted.
- As the heuristic ran for *n* steps, the final solution could be up to *n* swaps away from the original solution.

5.2.4 Simulated Annealing (SA)

An alternative to using Tabu search to widen the search area is to apply simulated annealing to the problem. In principle, SA could be able to produce near-optimal solutions for a problem, though usually at high processing cost. Applying a format similar that used in [Rabadi et al. 2002], in the course of the search, if there was an improvement in the cost of a neighboring schedule *S'* over the current schedule *S*, a transition from *S* to *S'* would automatically occur. If the cost of *S'* was higher than *S*, then the probability that the transition would occur was defined by :

$$prob_a = \exp\left(-\frac{difference\ in\ \cos t}{T}\right)$$

The pseudo code for the SA heuristic is shown in Fig. 7.

```
Set $S$ = schedule from initial $R_j$ (a priority job queue);
    T = START_TEMP;
    $cost_S$ = Cost of $S$;
    iter = 0;

While (iter < MAX_ITERS) and (successive iter without
transitions < MAX_STAGNANT_ITERS)
    Set trial = 0;
    While (trial < MAX_TRIALS) and (trials with improvement <
    MAX_SUCC_TRIALS)
        Generate schedule $S'$ (a neighbor to $S$);
        $cost_{S'}$ = Cost of $S'$;
        If ($cost_{S'}$ < $cost_S$) $cost_S$ = $cost_{S'}$ and $S$ = $S'$;
        Else ($cost_S$ = $cost_{S'}$ and $S$ = $S'$ with probability $prob_a$);
        trial++;
    iter++;
    T = TEMP_DECAY * T;
```

Fig. 7 Pseudo code for the simulated annealing heuristic

Some comments on the SA heuristic :

- The term “iterations” (represented by the variable *iter*)has been used for each step of the outer while-loop and “trials” (variable *trial*) for each step of the inner while-loop.
- Variables used in the heuristic are listed below. In general, high values for these variables improve solution quality but increase the computational cost.
    - *START_TEMP* is the starting temperature (T).
    - *TEMP_DECAY* $\in (0, 1)$ is the temperature decay rate. As T drops with each iteration, the heuristic becomes less likely to accept worse schedules than the current schedule *S* ($prob_a$ gets smaller).
    - *MAX_TRIALS* is the maximum number of trials that can be done at a single temperature T, even if no successful transitions occur.
    - *MAX_SUCC_TRIALS* is the maximum number of trials (transitions) resulting in a better schedule that can occur at one temperature T. If this is reached, the next iteration at a lower T starts immediately.
    - *MAX_ITERS* is the maximum number of different temperatures the heuristic iterates over, even if the schedule is still improving at the end.
    - *MAX_STAGNANT_ITERS* is the maximum number iterations (each at a different temperature) where no transitions from *S* to *S’* occur. If for *MAX_STAGNANT_ITERS* iterations, all *MAX_TRIALS* trials for the inner loop fail, then it is assumed that the probability of obtaining a transition with further iterations is extremely low, and thus the search is terminated.
- The priority job queue derived from the initial R_j values (as described earlier) was used as the starting point of the search and set to be *S*.

- To determine the cost of a given job queue, the adjusted greedy scheduling heuristic had to be run on the job queue. The sum of early and late penalties on the resulting schedule would be the cost of that job queue.
- As in Tabu Search, a neighbor was defined to be another job queue with 2 of the jobs' places in the original queue swapped.

5.2.5 Genetic Algorithm (GA)

Using a genetic algorithm could help to widen the search beyond the immediate vicinity of the initial solution created by the greedy priority ratings. In order to represent each possible solution as a chromosome which must be of a constant length, the *n*-length integer-array representation of the job priority queue was used.

One member of the initial population of valid solutions was always the array generated from the solution created by the greedy priority ratings. The rest of the members were created by a random ordering of all the non-breakdown jobs. The best and worst solutions of the entire population, together with their costs, were always noted. From this starting population of array solutions, the genetic algorithm heuristic shown in Fig. 8 was run.

```
Repeat GENERATIONS times
    Repeat (½ * POPULATION_SIZE) times
        Randomly select 2 parents from Pop ($Par_1$, $Par_2$);
        Create 2 offspring ($Off_1$ and $Off_2$) from $Par_1$ and $Par_2$ with a
        single crossover at a random point between $n/4$ and $3n/4$;
        Ensure $Off_1$ and $Off_2$'s arrays give legal solutions;
        $Off_1$ and $Off_2$ may be randomly mutated depending on mutation
        rate MUTATION;
        Repeat for $Off_1$ and $Off_2$
            If (cost of Off < $cost_{worst}$)
                Remove $array_{worst}$ from Pop;
                Add Off to Pop;
                Find the worst array solution in Pop, then set
                $array_{worst}$ and $cost_{worst}$;
                If (cost of Off < $cost_{best}$) {
                    $array_{best}$ = Off;
                    $cost_{best}$ = cost of $array_{best}$;
```

Fig. 8 Pseudo code for the genetic algorithm heuristic

Some comments on the GA heuristic :

- There are 3 variables that can be manipulated :
    - *POPULATION_SIZE* is the number of members in *Pop*. It is kept constant at all times as the worst members in *Pop* are progressively replaced by fitter offspring that have lower penalty costs.
    - *GENERATIONS* is used to dictate the length of time the heuristic is run for. In total, number of offspring generated throughout the heuristic = *POPULATION_SIZE* * *GENERATIONS*.
    - *MUTATION* is the chance that an offspring is changed from its initial configuration resulting from the crossover of its parents. When mutation occurs, the contents of 2 randomly-chosen cells in the array (neither of which have "-1" stored) are swapped.
- In a crossover operation, some value between $n/4$ and $3n/4$ is chosen as the crossover point. The values in the array of Off_1 are drawn from Par_1 before the

crossover point and from *Par*$_2$ after the crossover point. The reverse is true for *Off*$_2$. As long as all parents (starting from the initial population generation) have "-1" in the cells representing breakdown jobs, the offspring will also have "-1" in the same cells. The crossover operation is illustrated in Fig. 9.

- The crossover operation does not immediately guarantee a valid solution array, although all the breakdowns are correctly designated by "-1" in the offspring. There may be duplicates in the array (i.e. 2 jobs may both be designated the same ranking) due to the values inherited by the parents. To ensure legal offspring, the array must be checked so that no entries are repeated. Tied ranks are broken randomly.
- To calculate the penalty cost of a given solution array, the array must be converted to a priority job queue, which is then subjected to the adjusted greedy scheduling heuristic. The total costs of early and late penalties on the resulting schedule is the cost of the solution.

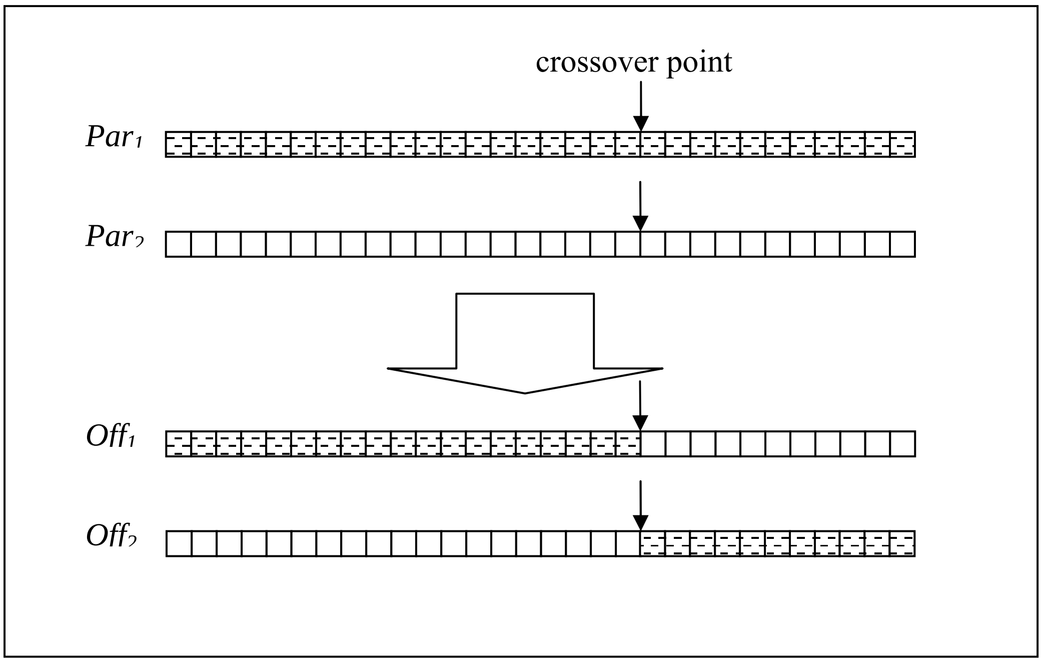

Fig. 9 Crossover operation for the genetic algorithm heuristic

5.2.6 Genetic Algorithm (GA) combined with Tabu Search (TS)

In difficult problems, the greedy scheduling heuristic is less likely to do well. A combination of GA search and TS could combine both elements of the GA's wide search for a reasonable approximate solution with the TS's search for a local best solution. Procedurally, it was simply a matter of running GA as described above, then running TS on the best result generated by the GA (instead of running the TS on the initial job queue based on starting R_j values).

Chapter 6

Experiments and Results

6.1 Generation of Test Problems

As there are no benchmark cases for this ETDDDsplit scheduling problem, a variety of test cases were generated. The distribution of the maintenance and breakdown jobs (m/b jobs) had to be hard-coded into the generation of test cases. As there obviously cannot be overlapping m/b jobs on the same machine, the m/b jobs had to be released in an approximately sequential manner.

For the non-breakdown and non-maintenance jobs (non-m/b jobs), the pattern of the release dates (a_j) could be subjected to some variation. Based on that variation, 4 groups of test cases were created (listed and explained below). In all cases, the release dates of the jobs had to be at least ($p_j + s_j$) before the end of the set schedule length. Other than that limitation, the release dates had no effect on the other characteristics of each non-m/b job. An example of a test case is included in Appendix A. (The following comments on the release dates of the jobs are applicable only to the non-m/b jobs) :

1. The release dates of the jobs were randomly spread out over the entire period being scheduled. If the ratio of the total schedule length to the number of jobs is less than about 6 : 1, then this case is more sparsely populated (hence easier to solve) than case (2), based the stated time intervals used in case (2).
2. The jobs were released singly at predictable time intervals of 3-10 time units apart. This tended to concentrate the jobs towards the start of the schedule.

3. The jobs' release dates followed a Gaussian distribution over the entire period being scheduled. The total schedule length corresponded to a normal distribution from -σ to σ (where σ = standard deviation of the curve), and the distribution corresponded to the probability that a job's release time would be set at a specific time in the schedule. Fig. 10 below highlights this concept. This distribution would spread out the jobs over the whole schedule, but with the centre part of the schedule having a greater load.
4. The jobs' release dates also followed a Gaussian distribution, but with the total schedule length corresponding to the normal distribution's range of -2σ to 2σ. This increased the concentration of jobs towards the centre of the schedule compared to case (3), and is also indicated in Fig. 10 below.

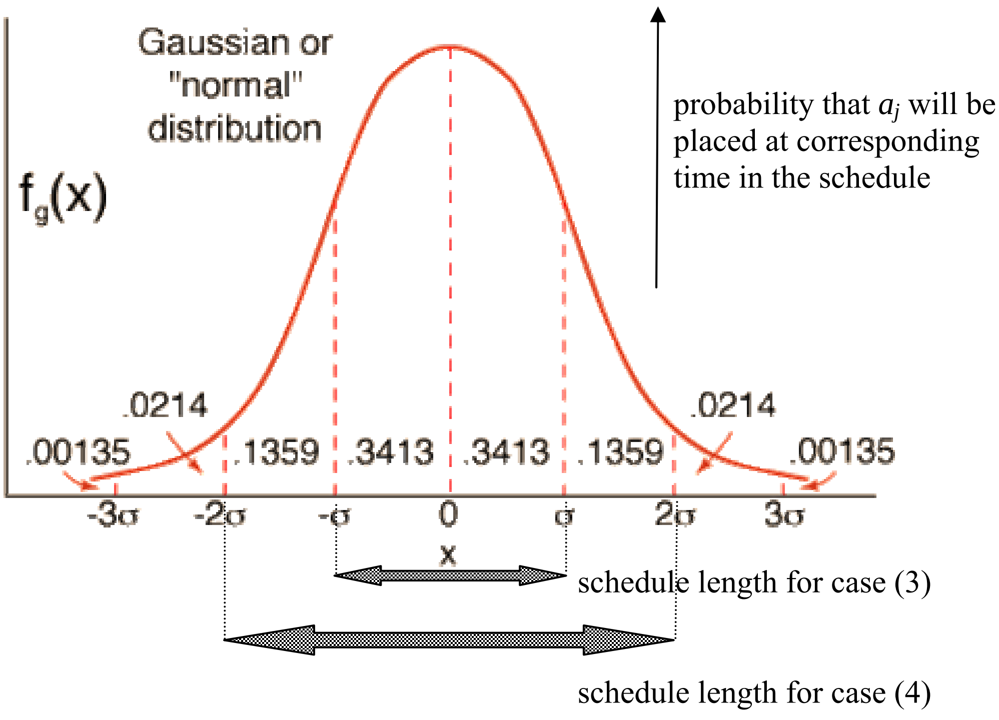
Gaussian or "normal" distribution
$f_g(x)$
probability that a_j will be placed at corresponding time in the schedule
.0214
.0214
.00135
.1359
.3413
.3413
.1359
.00135
-3σ
-2σ
-σ
0
σ
2σ
3σ
x
schedule length for case (3)
schedule length for case (4)

Fig. 10 Relationship between schedule length, a Gaussian distribution and jobs' release dates

The characteristics of the test cases generated were :

Global variables

- 5% of all jobs were maintenance jobs, a further 5% were breakdown jobs
- there were n jobs, $n = 200$
- there were m machines, $m = 5$
- the total period being scheduled was from time = 0 to time = 2000, for an overall schedule length to job ratio of slightly under 10 : 1
- $maxServer \in [1, 4]$

Machine variables

- $r_i \in [1, 5]$ for all machines M_i, i $\in \{1, \ldots, m\}$

Job variables (for non-maintenance and non-breakdown jobs) for job J_j, j $\in \{1, \ldots, n\}$

- setup time, $s_j \in [1, 5]$
- processing requirement, $p_j \in [20, 100]$
- time window, $d_j - a_j \in [200, 250]$
- early penalty, $e_j \in [5, 10]$
- late penalty, $t_j \in [50, 60]$
- maximum number of subsections allowed, $maxSplit_j \in [5, 10]$
- maximum number of simultaneous setups allowed, $maxParallel_j \in [1, 5]$

Job variables (for maintenance and breakdown jobs)

- setup time, s_{mj} or $s_{bj} = 0$
- processing requirement, p_{mj} or $p_{bj} \in [10, 40]$
- time window for
    - maintenance job, $d_{mj} - a_{mj} = 1.5 * p_{mj}$ (small time window)
    - breakdown job, $d_{bj} - a_{bj} = p_{bj}$ (exact time window)

- early penalty for
    - maintenance job, $e_{mj} = 0$
    - breakdown job, $e_{bj} = 1000\,000$
- late penalty, t_{mj} or $t_{bj} = 1000\,000$
- maximum subsections allowed, $maxSplit_{mj}$ or $maxSplit_{bj} = 1$
- maximum simultaneous setups allowed, $maxParallel_{mj}$ or $maxParallel_{bj} = 1$

10 instances of each test case were generated and the heuristics described in Chapter 5 were run on them. All experiments were run on a Pentium4 2.4GHz processor with 512MB RAM and coded using Java JDK 1.4. A sample of the result obtained using tabu search on the test case in Appendix A is given in Appendix B. The tabu search variables were chosen based on the results obtained in the tests described in section 6.2 below.

6.2 Heuristic Variable Settings

The tabu search, genetic algorithm and simulated annealing heuristics each involved a few variables which dictated the rate at which the solution space was explored. A few experiments varying the variables were run to determine which values would give a reasonable representation of the heuristic's performance. The eventual choice of values used to compare the heuristics' performances was based not only on the quality of the resulting solutions generated, but also on the additional amount of processing time required to return a significantly better result.

The scheduling problem used to test the variable values was one where the jobs' release dates followed a Gaussian distribution with schedule length corresponding to -2σ to 2σ of a normal curve. This problem was found to be moderately difficult and thus most suitable as a gauge of heuristic performance. Based on the results reported in section 6.3, the easier problems (with randomly spread job release dates and with the release dates following -σ to σ of a Gaussian distribution) would not sufficiently differentiate the quality of a heuristic, and the most difficult problem (with sequential release dates) made it too hard for any heuristic to create a good schedule.

6.2.1 Selection of Heuristic Variable Settings

The full experimental results for these tests are shown in Tables C1, C2 and C3 of Appendix C. The variable values used for each heuristic and the aggregate results are shown below in Tables 1, 2 and 3. All values in the tables represent the summed results of the 10 test cases run for each case, and all schedule penalty values shown are in thousands. The variable values shown the first row of each table were the values used to later compare the performance of the heuristics in section 6.3.

The results for the tabu search heuristic tests are shown below in Table 1. Each test was run 10 times with the listed values, and the best result returned as the solution. *TABU_TENURE* was set at 10, and each search was run using *n* (=200) iterations. The variables subject to change were *TS_NEIGHBOR*, *ALLOWANCE*, and *ITERATIONS*.

	TS_NEIGHBOR	*ALLOWANCE*	*ITERATIONS*	Schedule Penalty	Early Jobs	Tardy Jobs
1	100	1.1	100	198	143	23
2	300	1.1	100	189	138	24
3	50	1.1	100	246	142	32
4	100	1.2	100	282	175	35
5	100	1.1	200	186	149	22

Table 1 : Aggregated Experimental Results for Tabu Search

Compared with the top row, it is clear that increasing the size of the neighborhood searched (*TS_NEIGHBOR*) or increasing the breadth of the search (*ITERATIONS*) improved the results. However, the improvement was rather small in relation to additional amount of processing time required to create a solution schedule compared to the first case. Worse solution schedules clearly resulted from a smaller *TS_NEIGHBOR* which searched less of the neighborhood space at each step, and larger *ALLOWANCE*, which gave the heuristic too much leeway in selecting poorer solutions as the basis for the next step.

The results for the genetic algorithm tests are shown below in Table 2. Each test was run 10 times with the listed values and the best result returned as the solution. *POPULATION_SIZE* was kept constant at 100 for all tests. The variables subject to change were *GENERATIONS* and *MUTATION*.

	GENERATIONS	*MUTATION*	Schedule Penalty	Early Jobs	Tardy Jobs
1	100	0.15 (15% mutation rate)	159	176	13
2	100	0 (no mutation)	328	212	30
3	100	0.01	285	182	33
4	100	0.1	176	161	24
5	100	0.2	152	172	11
6	300	0.15	142	144	12

Table 2 : Aggregated Experimental Results for Genetic Algorithm

From these results, if the heuristic can search longer (larger *GENERATIONS*), a better results can be obtained. However, given the amount of additional processing time required for relatively small gain, it did not seem worthwhile to increase *GENERATIONS* for later comparisons. The *MUTATION* value had a strong effect on the solution quality, although even without any mutation, the crossover function did manage to improve the solution compared to the greedy and randomized greedy heuristics. (The values can be compared in Table 4.) To a point, higher mutation rates resulted in better solutions. However, the results when *MUTATION* = 0.15 and *MUTATION* = 0.2 were quite similar.

The results for the simulated annealing algorithm tests are shown below in Table 3. Following the rule-of-thumb applied by [Rabadi et al., 2002], *MAX_SUCC_TRIALS* = 0.1 * *MAX_TRIALS* and *MAX_STAGNANT_ITERS* = *MAX_ITERS* and were kept fixed at *MAX_TRIALS* = 1000, *MAX_SUCC_TRIALS* = 100, *MAX_ITERS* = 100000 and *MAX_STAGNANT_ITERS* = 10000. Each test was run 10 times with the listed values and the best result returned as the solution. The variables subject to change were *START_TEMP* and *TEMP_DECAY*.

	START_TEMP	*TEMP_DECAY*	Schedule Penalty	Early Jobs	Tardy Jobs
1	1000	0.99	386	196	43
2	700	0.99	437	205	43
3	1500	0.99	377	203	36
4	1000	0.95	421	193	38

Table 3 : Aggregated Experimental Results for Simulated Annealing

From these results, a faster cooling rate (smaller *TEMP_DECAY*) or a lower starting temperature (*START_TEMP*) generated a slightly worse solution schedule. On the other hand, starting with a higher value of *START_TEMP*, while probably better allowing the heuristic to leave any local minimum it might be trapped at to find a better solution elsewhere, took a much longer time to complete its search. Hence, the set of values in the first row of Table 3 were used for later tests.

6.3 Analysis of Heuristic Performances and Job Release Patterns

Once the values for the variables of the tabu search, genetic algorithm and simulated annealing heuristics had been fixed, attention could be focused on the relative performances of the heuristics with each other. The constants k_1 and k_2 used in the calculation of the jobs' priority ratings R_j were both set to 1, and the length of checktime set at 10. The heuristics and the final values for their associated variables were :

1. The greedy scheduling heuristic. (No variables, and it is only run once for each problem.)
2. The randomized scheduling heuristic. The modifier used to adjust the ratings R_j was a random value $\in [1, 2)$. This was run 1000 times for each test case, and the best result returned as the solution.
3. The tabu search heuristic was run 10 times with the variable values finalized in section 6.2 and the best result returned as the solution. The variable values were *TABU_TENURE* = 10, *ALLOWANCE* = 1.1 (10% allowance) and *TS_NEIGHBOR* = 100, with each search run using *n*(=200) iterations. Since a maximum of 100 neighborhoods were searched, the heuristic would search a maximum of 5% of the actual neighborhood for the next step ($\approx$ 0.5% of ${}^{n}C_2$ = 19900 actual neighbors) .
4. The simulated annealing heuristic was run 10 times with the variable values finalized in section 6.2 and the best result returned as the solution. The variable values were *START_TEMP* = 1000.0, *TEMP_DECAY* = 0.99, *MAX_TRIALS* = 1000, *MAX_SUCC_TRIALS* = 100, *MAX_ITERS* = 100000 and *MAX_STAGNANT_ITERS* = 10000.
5. The genetic algorithm heuristic was run 10 times with the variable values finalized in section 6.2 and the best result returned as the solution. The variable values used were *POPULATION_SIZE* = 100, *GENERATIONS* = 100 and *MUTATION* = 0.15 (15% mutation rate).
6. The genetic algorithm followed by the tabu search heuristic. This was run 10 times with the listed values and the best result returned as the solution. The variable values used were same as those used for the genetic algorithm alone and the tabu search heuristic alone : *POPULATION_SIZE* = 100,

GENERATIONS = 100, *MUTATION* = 0.15, *TS_NEIGHBOR* = 100, *TABU_TENURE* = 10 and *ALLOWANCE* = 1.1.

6.3.1 Experimental Results

The full experimental results are shown in Tables C4 to C7 of Appendix C. The aggregate results are shown below in Table 4. All values in the table represent the summed results of the 10 test cases generated for each test group and run by each heuristic.

	Greedy		Random Greedy		TS		SA		GA		GA+TS	
	Pen	E / T	Pen	E / T	Pen	E / T	Pen	E / T	Pen	E / T	Pen	E / T
Random	10	87 / 1	2	30 / 0	0	0 / 0	1	31 / 0	7	1 / 0	0	0 / 0
Gauss1	78	143 / 8	19	96 / 2	5	21 / 0	13	62 / 0	5	23 / 1	4	18 / 0
Gauss2	1038	194 / 80	585	185 / 57	198	143 / 23	386	196 / 43	159	176 / 13	156	153 / 17
Sequential	1942	199 / 141	806	240 / 53	128	146 / 20	573	221 / 36	182	165 / 20	117	158 / 13

Table 4 : Aggregated Experimental Results Comparing Heuristics

Notation for Table 2 :

Greedy	= the greedy scheduling heuristic
Random Greedy	= the randomized greedy scheduling heuristic
TS	= the tabu search heuristic
SA	= the simulated annealing heuristic
GA	= the genetic algorithm heuristic
GA+TS	= the genetic algorithm followed by tabu search heuristic
Random	= release dates were randomly spread out over the entire schedule
Gauss1	= release dates were spread in a normal distribution from -σ to σ
Gauss2	= release dates were spread in a normal distribution from -2σ to 2σ
Sequential	= release date were released at predictable intervals at the start of the schedule

E = total number of (non-m/b) jobs that completed before their due dates
T = total number of (non-m/b) jobs that completed after their due dates
Pen = total penalty of the solution schedule (values are in thousands)

6.3.2 Comparison of the Heuristics' Performances

In order of performance, with the heuristic that performed the best first :

1. The genetic algorithm followed by tabu search heuristic. (GA+TS)
2. The tabu search heuristic. (TS)
3. The genetic algorithm heuristic. (GA)
4. The simulated annealing heuristic. (SA)
5. The randomized greedy scheduling heuristic. (Random Greedy)
6. The greedy scheduling heuristic. (Greedy)

The fact that Random Greedy consistently out-performed Greedy indicates that the priority rating equation has room for improvement. On the other hand, since the modifier multiplier used is always a small value (between 1 and 2), it also implies that the priority rating has probably managed to correctly capture some aspect of the problem that determines which job should be scheduled first. The main effect of the random modifier was to allow the job order to be shifted slightly so that more jobs finished early rather than late, taking advantage of $e_j << t_j$. For instance, for the Sequential test group, the number of jobs that ended before their deadlines actually increased, but that was offset by the large drop in tardy jobs.

It is interesting that SA consistently performed worse than both TS and GA. In fact, there were several cases when even the Random Greedy heuristic did better than SA (e.g. Table C4 Case 10, Table C5 Case 9, Table C7 Case 3). This is despite the fact that processing time of SA at any of the variable settings tested was very much longer than any other heuristic, including GA+TS. There could be a few reasons for this :

- The SA variables have not been fully optimized for this problem. Although high values have been allocated to the variables, the heuristic does not converge to a very good solution. Despite the different starting temperatures and temperature decay rates used, a lot more time might would probably be needed to run the search, e.g. the rule-of-thumb applied by [Rabadi et al, 2002], *MAX_SUCC_TRIALS* = 0.1 * *MAX_TRIALS* and *MAX_STAGNANT_ITERS* = *MAX_ITERS*, might be inappropriate.
- The problem itself is not suitable for an SA approach. SA assumes that, like in the physical world, the global solution-landscape tends to "tilt" towards a global minimum. With this assumption, a form of very slow reverse-hill-climbing heuristic with sufficient allowance to jump over for local "bumps" should eventually reach the best solution. This analogy to the physics may not hold for the ETDDDsplit problem. Compared to SA, TS's allowance for "bumps" does not decrease over time, and GA's search can fluctuate wildly over the solution landscape due to the mutations. This could mean that the solution landscape is too flat for SA to work effectively.

The performance of GA+TS was somewhat better than the performance of TS alone, but not by very much. The GA alone performed quite similarly to TS alone. That both the wide search alone (GA) and narrow search alone (TS) performed similarly, but that the combination of both was only slightly better supports the theory that the global landscape is rather flat, and does not tend to tilt towards a global minimum (the optimum solution) when using this representation of the problem and solution neighborhoods. It also means that it could be very difficult to find the best global solution, as it could have very little relation to the quality of the solutions in its immediate neighborhood.

6.3.3 Comparison of the Test Groups

In order of difficulty, with the easiest test group first :

1. The release dates were randomly spread out over the entire schedule. (Random)
2. The release dates were spread out in a normal distribution from -σ to σ. (Gauss1)
3. The release dates were spread out in a normal distribution from -2σ to 2σ. (Gauss2)
4. The release date were released at predictable intervals at the start of the schedule. (Sequential)

The relative difficulty of Gauss2 and Sequential were about the same. Both had a congestion of jobs. Gauss2' congestion was at the centre of the schedule, and Sequential's was at the start of the schedule. A congestion at the start of the schedule would obviously increase the overall tardiness of the jobs, driving up the total schedule penalties, even the sum of the absolute differences between the deadlines and actual completion times were similar for both cases. For both Gauss2 and Sequential, the congestion of jobs would logically make it more difficult to generate a good solution, no matter what heuristic was used. The best results occurred when the jobs are evenly spread out over the scheduled time period.

Chapter 7

Conclusions and Future Work

In this thesis, we introduced a new NP-complete scheduling problem, the early-tardy distinct due date machine scheduling problem with splitting jobs and setup times (ETDDDsplit). Although it is not yet well-researched, it has practical applications in the manufacturing industry. This problem carries some characteristics from both the standard early-tardy machine scheduling problem, and from the scheduling problem with lotsizing or job splitting.

We have successfully adapted 3 of the standard search heuristics to the ETDDDsplit problem. A priority rating is first used to rate the jobs in order of urgency, which a greedy heuristic can then used to create an initial solution. Tabu search, genetic algorithms and simulated annealing can be applied onto the job orderings and a heuristic used to generate reasonable solutions for problems of moderate size. The results show that the solution space does not easily lend itself to any hill-climbing approach to finding the optimal solution schedule. The characteristics of the job release dates have also been found to have a strong effect on the quality of the solution schedules that can be found. In terms of real life situations, this means that some care needs to be taken in accepting jobs for processing, based on the jobs that have already been accepted or are predicted to be offered to the factory work floor.

There is much room for further research in the ETDDDsplit problem. For instance, the model representing the problem could be further refined by making e_j nonconstant. It could instead reflect the incremental storage cost of partially completed orders, by being proportional to the size of the completed subsections. Thus, an early penalty would be applied to each subsection as it is completed instead of only being applied to the completed job.

The work here represents one possible way break down the problem into 2 distinct phases, the job ordering phase and the scheduling phase to insert idle machine time. Other ways to order, schedule and split the jobs should also be examined. The conclusions drawn here about the difficulty of the problem and the topology of the solution space may be changed by a different representation of the problem and its potential solutions. Different search or AI heuristics could also be tested to see if better solutions emerge. Some were highlighted in Chapter 4, but other possibilities include Ant Colony and “Squeaky Wheel” optimizations.

References

K.R.Baker, G.D.Scudder, (1990) “Sequencing with Earliness and Tardiness Penalties: A Review”, Operations Research 38 (1) pp22-36

B.Chen, (1993) “A better heuristic for preemptive parallel machine scheduling with setup times”, SIAM Journal on Computing 22, pp1303-1318

B.Chen, C.N.Potts, G.J.Woeginger, (1998) “A review of machine scheduling : complexity, algorithms and approximability”, “Handbook of Combinatorial Optimization”, Kluwer, Dordretch, pp21-169

Z.L.Chen, W.B.Powell, (1995) “Solving Parallel Machine Scheduling Problems by Column Generation”, Technical Report, Statistics and Operation Research, Princeton University

H.Emmons, (1987) “Scheduling to a common due date on parallel uniform processors”, Naval Research Logistics 34, pp803-810

G.Finke, V.Gordon, J.M.Proth, (2002) “Scheduling with due dates (Annotated bibliography of complexity and algorithms)”, available from http://www-leibniz.imag.fr/LEIBNIZ/LesCahiers/2002/Cahier42/ResumCahier42.html

T.D.Fry, R.D.Armstrong, J.H.Blackstone, (1987) "Minimizing Weighted Absolute Deviation in Single Machine Scheduling" IIE Transactions 19, pp445-450

M.R.Garey, R.L.Graham, D.S.Johnson, (1978) "Performance Guarantees for Scheduling Algorithms", Operations Research 26, pp3-21

M.R.Garey, D.S.Johnson, (1979) "Computers and Intractability, A Guide to the Theory of NP-Completeness", Murray Hill

F.Glover, (1989) "Tabu Search - Part I", ORSA Journal on Computing 1 (3), pp190-206

F.Glover, (1990) "Tabu Search - Part II", ORSA Journal on Computing 2 (1), pp4-32

A.Gozzi, M.Paolucci, A.Boccalatte, (2002) "Autonomous Agents Applied to Manufacturing Scheduling Problems : A Negotiation-Based Heuristic Approach" in V.Marik, O.Stepankova, H.Krautwurmova, M.luck (Eds.): Multi-Agent Systems and Application II, Selected Revised Papers: 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001, LNAI 2322, Springer Verlag, pp194-203

R.L.Graham, E.L.Lawler, J.K.Lenstra, A.H.G.Rinnooy Kan, (1979) "Optimization and approximation in deterministic sequencing and scheduling: a survey", Annals of Discrete Mathematics 5, pp287-326

A.Hamad, B.Sanugi, S.Salleh, (2002) "A Neural Network Approach for Distinct Due Date Job Scheduling Problems on Parallel Identical Machines", Journal of Theoretics 4 No.3

J.H.Holland, (1992) "Adaptation in natural and artificial systems" 2nd Ed., Cambridge MA : The MIT Press

J.A.Hoogeveen, S.L.van de Velde, (1996) "A Branch-and-Bound Algorithm for Single-Machine Earliness-Tardiness Scheduling with Idle Time", INFORMS Journal on Computing 8, pp402-412

A.Jain, S.Meeran, (1998) "Job-Shop Scheduling Using Neural Networks", International Journal of Production Research 36 (5), pp1249-1272

R.J.W.James, J.T.Buchanan, (1997) "A neighbourhood scheme with a compressed solution space for the early/tardy scheduling problem", European Journal of Operational Research 102, pp513-527

R.James, (1998) "Long Term Memory Strategies for Solving the Early/Tardy Scheduling Problem", available from http://citeseer.nj.nec.com/367151.html

S.Kirkpatrick, C.D.Gelatt Jr., M.P.Vecchi, (1983) "Optimization by Simulated Annealing", Science 220, pp671-680

W.Kubiak, S.Lou, R.Sethi, (1990) “Equivalence of mean flow time problems and mean absolute deviation problems”, Operations Research Letters 9, pp37-374

E.L.Lawler, J.K.Lenstra, A.H.G.Rinnooy Kan, D.B.Shmoys, (1993) “Sequencing and scheduling: algorithms and complexity”, In Handbooks in operations research and management science (4), Amsterdam, pp445-522

C.L.Monma, C.N.Potts, (1993) “Analysis of heuristics for preemptive parallel machine scheduling with job setup times”, Operations Research 41, pp981-993

P.S.Ow, T.W.Morton, (1989) “The Single-Machine Early/Tardy Problem”, Management Science 35, pp177-191

C.N.Potts, L.N.Van Wassenhove, (1992) “Integrating scheduling with batching and lotsizing: a review of algorithms and complexity”, Journal of the Operational Research Society 43 (5) , pp395-406

G.Rabadi, G.Anagnostopoulos, M.Mollaghasemi, (2002) “A Simulated Annealing Algorithm for a Scheduling Problem with Setup Times”, Proc. of The Industrial Engineering Research Conference, Florida

C.R.Reeves, (1993) “Improving the Efficiency of Tabu Search for Machine Scheduling Problems”, Journal of the Operational Research Society 44, pp375-392

P.Serafini, (1996) “Scheduling jobs on several machines with the job splitting property”, Operations Research 44 (4), pp617-628

F.Sivrikaya-Serifoglu, G.Ulusoy, (1999) "Parallel machine scheduling with earliness and tardiness penalties", Computers and Operations Research 26, pp773-787

J.M.S.Valente, R.A.F.S.Alves, (2003) “Improved Heuristics for the Early/Tardy Scheduling Problem with No Idle Time”, working paper from www.fep.up.pt/investigacao/workingpapers/wp126.pdf

W.Xing, J.Zhang, (1998) “Splitting parallel machine scheduling”, Operations Research Tradnsactions 2, pp30-41

W.Xing, J.Zhang, (2000) “Parallel machine scheduling with splitting jobs”, Discrete Applied Mathematics 103, pp259-269

C.A.Yano, Y.D.Kim, (1991) “Algorithms for a class of single-machine weighted tardiness and earliness problems”, European Journal of Operational Research 52, pp167-178

Appendices

Appendix A Sample Test Case

A sample test case is shown below. In this test case, the jobs were released at time intervals of 3-10 time units. The other variables are described in Chapter 6.

Global variables

```
#max_server|schedule_length
#max_server: maximum number of parallel setups for any job (int)
#schedule length: total length of schedule (int)
4|2000
```

Machine Variables

```
#mid|mname|rate
#mid: Machine ID (int)
#mname: Machine name (string)
#rate: Workrate of machine (int)
1|Machine 1|2
2|Machine 2|2
3|Machine 3|2
4|Machine 4|4
5|Machine 5|3
```

Job Variables

```
#jid|jname|setup_time|release_time|deadline|size|early_pen|late_pen|max_split|max_para
llel|paying_job|mid
#jid: Job ID (int)
#jname: Job name (string)
#setup_time: Setup time in hours (int)
#release_time: Release time of job (int) - when the job can be started
#deadline: Latest end time without penalty (int)
#size: Number of time units required to complete the job at workrate 1 (int) -
significant only for paying jobs|||||||||||
#early_pen: Penalty for early completion in cost per hour (int)
#late_pen: Penalty for tardiness in cost per hour (int)
#max_split: Maximum number of sections for this job can be split into (int)
"#max_parallel: Maximum number of sections of this job that can run concurrently, also
affects for setup time (int)"
#paying_job : Whether machine workrate is a factor for this job (Y/N) - N implies
maintenance/breakdown
"#mid : is paying_job=N, the required Machine ID, else -1"

1|Job 1|2|6|218|82|7|51|5|3|Y|-1
2|BreakDown 1|0|16|45|29|1000000|1000000|1|1|N|5
3|Job 2|1|21|221|33|5|52|5|3|Y|-1
4|Job 3|1|25|228|95|7|58|9|3|Y|-1
5|Job 4|3|31|255|71|6|52|6|4|Y|-1
6|Job 5|3|38|258|87|8|60|10|1|Y|-1
7|Job 6|5|46|288|94|7|50|8|4|Y|-1
8|Job 7|5|50|254|27|10|59|6|5|Y|-1
9|Job 8|5|59|268|88|10|59|9|4|Y|-1
10|Job 9|4|65|291|28|7|60|9|2|Y|-1
11|Job 10|5|70|301|61|6|57|7|3|Y|-1
12|Job 11|4|79|315|40|10|57|8|1|Y|-1
13|Job 12|5|86|299|66|10|52|5|4|Y|-1
14|Job 13|3|92|324|20|10|54|10|5|Y|-1
15|Job 14|1|97|297|85|5|60|6|3|Y|-1
16|Job 15|2|105|328|72|9|51|7|4|Y|-1
17|Job 16|2|110|354|26|9|51|5|4|Y|-1
```

```
18|Job 17|2|117|366|98|8|57|9|3|Y|-1
19|Job 18|4|120|368|90|10|56|5|4|Y|-1
20|Job 19|3|123|346|24|6|56|6|4|Y|-1
21|Job 20|2|133|380|88|8|50|9|3|Y|-1
22|Job 21|5|136|379|51|8|58|9|5|Y|-1
23|Maintenance 1|0|135|184|33|0|1000000|1|1|N|3
24|Job 22|3|149|372|68|5|51|5|2|Y|-1
25|Job 23|4|154|364|45|6|60|7|4|Y|-1
26|Job 24|4|164|412|35|9|56|7|5|Y|-1
27|Job 25|2|167|412|75|5|60|8|1|Y|-1
28|Job 26|4|177|421|42|7|52|6|5|Y|-1
29|Job 27|4|182|427|71|6|58|9|5|Y|-1
30|Job 28|2|192|397|93|8|54|6|1|Y|-1
31|Job 29|1|201|437|45|9|57|8|1|Y|-1
32|Job 30|1|204|418|77|8|57|10|4|Y|-1
33|Job 31|5|208|448|26|9|56|10|4|Y|-1
34|Job 32|2|217|434|20|8|55|8|1|Y|-1
35|Job 33|3|222|430|23|10|58|5|3|Y|-1
36|Maintenance 2|0|224|262|26|0|1000000|1|1|N|4
37|BreakDown 2|0|238|253|15|1000000|1000000|1|1|N|1
38|Job 34|5|241|481|81|7|59|7|3|Y|-1
39|Job 35|1|250|451|43|7|56|8|5|Y|-1
40|Job 36|3|260|492|21|5|56|6|3|Y|-1
41|Job 37|3|263|491|42|7|56|7|3|Y|-1
42|Job 38|1|266|476|69|5|52|6|4|Y|-1
43|Job 39|5|270|511|48|6|53|8|2|Y|-1
44|Job 40|1|277|498|60|5|53|5|2|Y|-1
45|Job 41|4|281|505|71|10|58|7|4|Y|-1
46|Job 42|1|291|510|34|9|53|9|3|Y|-1
47|Job 43|2|301|531|26|9|57|9|5|Y|-1
48|Job 44|4|305|515|66|8|51|8|1|Y|-1
49|Job 45|4|311|528|85|10|53|7|5|Y|-1
50|Job 46|1|317|523|99|9|54|6|5|Y|-1
51|Job 47|5|324|541|36|6|51|8|4|Y|-1
52|Job 48|1|331|541|21|7|59|7|1|Y|-1
53|Job 49|5|336|539|59|6|59|5|3|Y|-1
54|Job 50|1|340|551|92|5|60|6|5|Y|-1
55|BreakDown 3|0|344|383|39|1000000|1000000|1|1|N|2
56|Job 51|4|354|562|32|6|52|6|5|Y|-1
57|Job 52|4|361|569|29|9|50|10|3|Y|-1
58|Job 53|5|368|571|54|8|53|6|1|Y|-1
59|Job 54|4|371|576|37|7|59|7|3|Y|-1
60|Job 55|5|378|613|60|5|57|8|4|Y|-1
61|Job 56|5|388|625|28|9|50|8|3|Y|-1
62|Job 57|2|391|621|35|8|55|8|4|Y|-1
63|Job 58|5|394|618|88|8|60|5|4|Y|-1
64|Job 59|1|403|615|74|9|58|6|4|Y|-1
65|Job 60|4|406|607|23|9|52|7|3|Y|-1
66|Job 61|4|413|626|97|9|50|8|1|Y|-1
67|Maintenance 3|0|414|457|29|0|1000000|1|1|N|2
68|Job 62|4|427|654|84|9|55|9|1|Y|-1
69|Job 63|5|430|675|52|6|50|10|5|Y|-1
70|Maintenance 4|0|424|478|36|0|1000000|1|1|N|3
71|Job 64|3|436|675|92|6|58|6|4|Y|-1
72|Job 65|1|439|680|29|9|53|8|1|Y|-1
73|Job 66|3|446|651|69|5|53|8|2|Y|-1
74|Job 67|3|456|669|92|6|50|6|5|Y|-1
75|Job 68|2|461|691|64|7|55|6|2|Y|-1
76|Job 69|1|471|705|69|10|51|5|3|Y|-1
77|Job 70|5|478|684|70|6|56|7|5|Y|-1
78|BreakDown 4|0|488|503|15|1000000|1000000|1|1|N|2
79|BreakDown 5|0|493|529|36|1000000|1000000|1|1|N|1
80|Job 71|5|498|734|64|6|53|5|4|Y|-1
81|Job 72|4|502|732|22|10|56|6|2|Y|-1
82|Job 73|4|511|750|35|8|54|10|1|Y|-1
83|Maintenance 5|0|507|556|33|0|1000000|1|1|N|5
84|Job 74|2|520|727|97|6|57|8|2|Y|-1
85|Job 75|2|526|760|65|10|50|7|3|Y|-1
86|Job 76|4|534|734|59|8|54|6|1|Y|-1
87|Job 77|5|542|747|23|6|53|10|4|Y|-1
88|Job 78|1|548|772|51|9|59|7|3|Y|-1
89|Job 79|4|558|766|73|10|57|9|1|Y|-1
90|Job 80|5|564|786|96|7|50|5|3|Y|-1
91|Job 81|3|574|802|75|8|55|9|2|Y|-1
92|Job 82|5|582|832|94|5|56|8|5|Y|-1
```

```
93|Job 83|5|590|791|59|7|52|9|5|Y|-1
94|Job 84|2|596|831|42|8|54|8|4|Y|-1
95|Job 85|4|606|808|62|7|54|10|2|Y|-1
96|Job 86|5|613|842|43|10|51|6|5|Y|-1
97|Job 87|5|619|853|61|5|59|8|5|Y|-1
98|Job 88|3|624|854|24|8|59|6|3|Y|-1
99|Job 89|4|629|874|55|5|55|6|5|Y|-1
100|Maintenance 6|0|635|661|18|0|1000000|1|1|N|1
101|Job 90|1|649|898|33|5|57|7|2|Y|-1
102|Job 91|1|655|860|71|6|50|8|3|Y|-1
103|Job 92|5|660|887|71|7|52|5|2|Y|-1
104|Job 93|1|664|864|38|6|59|7|1|Y|-1
105|Job 94|1|668|903|88|5|55|6|5|Y|-1
106|Job 95|3|674|903|86|7|56|8|2|Y|-1
107|Job 96|3|677|916|80|10|51|10|1|Y|-1
108|Job 97|5|685|898|49|7|53|7|1|Y|-1
109|Job 98|1|693|940|73|5|59|10|1|Y|-1
110|Job 99|3|700|907|60|8|56|8|3|Y|-1
111|Job 100|1|704|918|72|9|57|7|1|Y|-1
112|Job 101|5|707|938|34|5|55|6|4|Y|-1
113|Job 102|5|716|949|61|8|53|6|4|Y|-1
114|Job 103|2|724|944|83|5|52|8|3|Y|-1
115|Job 104|5|727|927|24|7|50|9|3|Y|-1
116|BreakDown 6|0|731|748|17|1000000|1000000|1|1|N|4
117|Job 105|2|734|960|89|5|56|7|1|Y|-1
118|Job 106|2|744|951|22|6|53|9|5|Y|-1
119|BreakDown 7|0|750|782|32|1000000|1000000|1|1|N|3
120|Job 107|5|755|1003|86|6|58|9|1|Y|-1
121|Job 108|1|761|979|92|9|53|5|5|Y|-1
122|Job 109|1|771|1006|30|8|60|9|4|Y|-1
123|Job 110|4|775|1002|85|10|60|5|3|Y|-1
124|Job 111|1|785|1019|66|5|51|7|1|Y|-1
125|Job 112|5|788|989|59|5|57|5|3|Y|-1
126|Job 113|4|794|1026|25|7|51|8|2|Y|-1
127|Job 114|1|797|1028|64|7|53|8|3|Y|-1
128|Job 115|1|802|1039|80|8|54|7|4|Y|-1
129|Job 116|1|811|1026|86|9|57|10|3|Y|-1
130|BreakDown 8|0|819|834|15|1000000|1000000|1|1|N|3
131|Job 117|3|829|1048|97|10|52|7|2|Y|-1
132|Job 118|5|833|1057|23|9|51|7|4|Y|-1
133|Job 119|2|840|1078|84|10|57|5|5|Y|-1
134|Job 120|5|850|1080|36|6|53|9|1|Y|-1
135|Job 121|4|855|1099|24|6|55|6|4|Y|-1
136|Job 122|4|860|1072|74|9|52|10|3|Y|-1
137|Job 123|4|864|1112|46|9|51|7|3|Y|-1
138|Maintenance 7|0|868|888|14|0|1000000|1|1|N|4
139|Maintenance 8|0|869|905|24|0|1000000|1|1|N|5
140|Job 124|4|885|1135|26|6|58|9|4|Y|-1
141|Job 125|1|890|1122|24|5|56|7|1|Y|-1
142|Job 126|3|896|1128|53|9|59|10|3|Y|-1
143|Job 127|4|902|1139|88|8|57|6|3|Y|-1
144|Job 128|5|910|1117|74|8|57|6|1|Y|-1
145|Job 129|2|917|1142|71|9|60|5|1|Y|-1
146|Job 130|1|922|1132|56|5|51|5|5|Y|-1
147|Job 131|5|926|1138|69|10|53|5|1|Y|-1
148|Job 132|3|929|1134|35|7|50|8|5|Y|-1
149|Job 133|4|937|1146|61|8|60|5|1|Y|-1
150|Job 134|3|944|1192|87|7|51|8|3|Y|-1
151|Job 135|3|949|1194|42|10|58|7|1|Y|-1
152|Job 136|1|959|1195|63|5|58|5|1|Y|-1
153|Job 137|3|963|1163|67|7|52|7|4|Y|-1
154|Job 138|2|972|1212|80|5|50|6|1|Y|-1
155|Job 139|5|975|1181|28|8|50|5|3|Y|-1
156|Job 140|5|979|1203|89|10|50|10|1|Y|-1
157|Maintenance 9|0|983|1007|16|0|1000000|1|1|N|2
158|Job 141|1|996|1241|85|9|59|7|4|Y|-1
159|Job 142|3|999|1206|78|10|59|8|3|Y|-1
160|Job 143|2|1007|1219|97|9|53|6|4|Y|-1
161|Job 144|5|1013|1252|26|6|56|10|3|Y|-1
162|Job 145|5|1021|1227|52|5|56|9|1|Y|-1
163|Job 146|4|1031|1278|35|9|53|10|3|Y|-1
164|Job 147|2|1039|1284|28|6|51|7|3|Y|-1
165|Maintenance 10|0|1045|1060|11|0|1000000|1|1|N|3
166|Job 148|1|1053|1275|43|7|54|9|1|Y|-1
167|Job 149|1|1063|1300|32|6|58|6|5|Y|-1
```

```
168|Job 150|2|1072|1321|66|9|50|7|3|Y|-1
169|Job 151|3|1078|1313|27|10|60|5|4|Y|-1
170|BreakDown 9|0|1085|1101|16|1000000|1000000|1|1|N|5
171|Job 152|2|1089|1310|80|7|55|9|5|Y|-1
172|Job 153|1|1092|1337|20|5|59|9|2|Y|-1
173|Job 154|1|1098|1326|73|6|54|5|5|Y|-1
174|Job 155|3|1105|1343|26|8|53|9|4|Y|-1
175|Job 156|4|1108|1339|69|10|56|8|5|Y|-1
176|Job 157|5|1116|1319|48|9|58|6|1|Y|-1
177|Job 158|3|1120|1366|49|8|57|9|4|Y|-1
178|Job 159|1|1128|1362|38|9|60|10|2|Y|-1
179|Job 160|5|1133|1379|41|9|52|9|3|Y|-1
180|Job 161|1|1137|1358|43|10|57|6|4|Y|-1
181|Job 162|4|1145|1375|86|10|56|10|2|Y|-1
182|Job 163|5|1152|1359|91|6|50|6|2|Y|-1
183|Job 164|2|1155|1388|25|10|54|9|3|Y|-1
184|Job 165|2|1161|1390|45|9|58|8|5|Y|-1
185|BreakDown 10|0|1165|1190|25|1000000|1000000|1|1|N|3
186|Job 166|4|1174|1407|61|6|51|10|3|Y|-1
187|Job 167|2|1180|1386|47|9|56|5|5|Y|-1
188|Job 168|3|1188|1426|85|9|51|10|5|Y|-1
189|Job 169|1|1198|1424|83|10|56|9|5|Y|-1
190|Job 170|1|1206|1410|74|9|58|5|3|Y|-1
191|Job 171|1|1216|1421|33|10|55|7|2|Y|-1
192|Job 172|4|1221|1470|35|10|57|10|2|Y|-1
193|Job 173|2|1225|1450|84|7|55|6|5|Y|-1
194|Job 174|1|1235|1446|93|8|51|5|3|Y|-1
195|BreakDown 11|0|1245|1263|18|1000000|1000000|1|1|N|2
196|Job 175|2|1251|1488|38|6|51|10|3|Y|-1
197|Job 176|1|1255|1472|63|8|50|8|3|Y|-1
198|Job 177|5|1263|1465|65|6|60|8|4|Y|-1
199|Job 178|2|1268|1516|27|7|59|10|3|Y|-1
200|Maintenance 11|0|1268|1287|13|0|1000000|1|1|N|1
```

Appendix B Sample Schedule Solution

A sample schedule solution is shown below. This is the result of a tabu search (without the genetic algorithm) on the sample test case given in Appendix A. The variable values used are given in section 6.2.1, first row of Table 1.

Schedule for Machine 1

```
Schedule for Machine 1
Job 2 Section 2 206 to 221 with setup time of 1
BreakDown 2 Section 1 238 to 253 with setup time of 0
Job 6 Section 1 253 to 288 with setup time of 5
Job 10 Section 2 288 to 301 with setup time of 5
Job 28 Section 3 309 to 327 with setup time of 2
Job 22 Section 3 327 to 339 with setup time of 3
Job 16 Section 1 339 to 354 with setup time of 2
Job 23 Section 1 354 to 364 with setup time of 4
Job 22 Section 1 364 to 372 with setup time of 3
Job 20 Section 3 372 to 380 with setup time of 2
Job 25 Section 1 380 to 412 with setup time of 2
Job 33 Section 1 415 to 430 with setup time of 3
Job 49 Section 3 430 to 440 with setup time of 5
Job 38 Section 1 440 to 476 with setup time of 1
Job 46 Section 5 476 to 478 with setup time of 1
Job 36 Section 1 478 to 492 with setup time of 3
BreakDown 5 Section 1 493 to 529 with setup time of 0
Job 50 Section 4 529 to 534 with setup time of 1
Job 47 Section 2 534 to 541 with setup time of 5
Job 51 Section 1 542 to 562 with setup time of 4
Job 52 Section 2 564 to 569 with setup time of 4
Job 54 Section 3 569 to 576 with setup time of 4
Job 61 Section 3 576 to 606 with setup time of 4
Job 56 Section 1 606 to 625 with setup time of 5
Job 68 Section 4 625 to 643 with setup time of 2
Maintenance 6 Section 1 643 to 661 with setup time of 0
Job 67 Section 2 661 to 668 with setup time of 3
Job 63 Section 2 668 to 675 with setup time of 5
Job 70 Section 2 675 to 684 with setup time of 5
Job 68 Section 2 684 to 691 with setup time of 2
Job 76 Section 1 700 to 734 with setup time of 4
Job 77 Section 2 734 to 740 with setup time of 5
Job 75 Section 2 740 to 760 with setup time of 2
Job 80 Section 3 761 to 777 with setup time of 5
Job 83 Section 2 777 to 791 with setup time of 5
Job 85 Section 3 791 to 802 with setup time of 4
Job 82 Section 1 802 to 832 with setup time of 5
Job 87 Section 1 832 to 853 with setup time of 5
Job 94 Section 4 853 to 857 with setup time of 1
Job 95 Section 1 857 to 903 with setup time of 3
Job 100 Section 3 903 to 905 with setup time of 1
Job 96 Section 1 905 to 916 with setup time of 3
Job 101 Section 1 916 to 938 with setup time of 5
Job 106 Section 1 938 to 951 with setup time of 2
Job 107 Section 2 959 to 979 with setup time of 5
Job 111 Section 3 979 to 998 with setup time of 1
Job 115 Section 1 998 to 1039 with setup time of 1
Job 118 Section 1 1040 to 1057 with setup time of 5
Job 122 Section 2 1061 to 1072 with setup time of 4
Job 130 Section 4 1074 to 1083 with setup time of 1
Job 121 Section 1 1083 to 1099 with setup time of 4
Job 126 Section 2 1099 to 1114 with setup time of 3
Job 133 Section 2 1114 to 1135 with setup time of 4
Job 129 Section 2 1135 to 1138 with setup time of 2
Job 136 Section 3 1153 to 1170 with setup time of 1
Job 135 Section 1 1170 to 1194 with setup time of 3
Job 140 Section 1 1194 to 1203 with setup time of 5
Job 138 Section 2 1203 to 1206 with setup time of 2
Job 143 Section 1 1206 to 1219 with setup time of 2
Maintenance 11 Section 1 1274 to 1287 with setup time of 0
Job 151 Section 1 1296 to 1313 with setup time of 3
Job 157 Section 1 1313 to 1319 with setup time of 5
Job 156 Section 1 1319 to 1339 with setup time of 4
Job 159 Section 1 1342 to 1362 with setup time of 1
Job 158 Section 1 1362 to 1366 with setup time of 3
Job 166 Section 1 1372 to 1407 with setup time of 4
Job 170 Section 1 1407 to 1410 with setup time of 1
Job 171 Section 1 1410 to 1421 with setup time of 1
Job 174 Section 2 1427 to 1446 with setup time of 1
Job 177 Section 1 1446 to 1465 with setup time of 5
```

Schedule for Machine 2

```
Schedule for Machine 2
Job 5 Section 2 209 to 254 with setup time of 3
Job 8 Section 4 254 to 266 with setup time of 5
Job 6 Section 2 266 to 288 with setup time of 5
Job 10 Section 3 288 to 301 with setup time of 5
Job 23 Section 2 323 to 344 with setup time of 4
BreakDown 3 Section 1 344 to 383 with setup time of 0
Job 27 Section 1 387 to 427 with setup time of 4
Maintenance 3 Section 1 428 to 457 with setup time of 0
Job 50 Section 5 457 to 459 with setup time of 1
Job 44 Section 5 459 to 488 with setup time of 4
BreakDown 4 Section 1 488 to 503 with setup time of 0
Job 44 Section 2 503 to 510 with setup time of 4
Job 46 Section 2 510 to 519 with setup time of 1
Job 45 Section 2 519 to 528 with setup time of 4
Job 48 Section 1 529 to 541 with setup time of 1
Job 50 Section 2 541 to 551 with setup time of 1
Job 54 Section 4 558 to 571 with setup time of 4
Job 61 Section 4 571 to 576 with setup time of 4
Job 59 Section 1 577 to 615 with setup time of 1
Job 66 Section 1 615 to 651 with setup time of 3
Job 67 Section 3 651 to 667 with setup time of 3
Job 65 Section 2 667 to 675 with setup time of 1
Job 70 Section 3 675 to 684 with setup time of 5
Job 74 Section 2 716 to 727 with setup time of 2
Job 73 Section 1 728 to 750 with setup time of 4
Job 80 Section 1 750 to 786 with setup time of 5
Job 85 Section 4 786 to 791 with setup time of 4
Job 82 Section 2 805 to 832 with setup time of 5
Job 87 Section 2 833 to 853 with setup time of 5
Job 93 Section 2 853 to 860 with setup time of 1
Job 94 Section 3 860 to 868 with setup time of 1
Job 97 Section 1 868 to 898 with setup time of 5
Job 94 Section 1 898 to 903 with setup time of 1
Job 102 Section 1 913 to 949 with setup time of 5
Job 111 Section 4 976 to 979 with setup time of 1
Job 112 Section 1 979 to 989 with setup time of 5
Maintenance 9 Section 1 991 to 1007 with setup time of 0
Job 113 Section 1 1009 to 1026 with setup time of 4
Job 129 Section 5 1038 to 1053 with setup time of 2
Job 120 Section 2 1053 to 1072 with setup time of 5
Job 129 Section 4 1072 to 1091 with setup time of 2
Job 127 Section 1 1091 to 1139 with setup time of 4
Job 140 Section 2 1148 to 1194 with setup time of 5
Job 138 Section 3 1194 to 1203 with setup time of 2
Job 143 Section 2 1203 to 1219 with setup time of 2
BreakDown 11 Section 1 1245 to 1263 with setup time of 0
Job 152 Section 1 1268 to 1310 with setup time of 2
Job 156 Section 2 1316 to 1339 with setup time of 4
Job 158 Section 2 1339 to 1366 with setup time of 3
Job 167 Section 2 1368 to 1386 with setup time of 2
Job 170 Section 2 1386 to 1410 with setup time of 1
Job 171 Section 2 1413 to 1421 with setup time of 1
Job 177 Section 2 1441 to 1465 with setup time of 5
```

Schedule for Machine 3

```
Schedule for Machine 3
Maintenance 1 Section 1 151 to 184 with setup time of 0
Job 4 Section 1 216 to 255 with setup time of 3
Job 8 Section 3 255 to 268 with setup time of 5
Job 14 Section 2 268 to 279 with setup time of 1
Job 10 Section 4 279 to 291 with setup time of 5
Job 14 Section 1 291 to 297 with setup time of 1
Job 25 Section 3 320 to 328 with setup time of 2
Job 28 Section 2 328 to 331 with setup time of 2
Job 19 Section 1 331 to 346 with setup time of 3
Job 21 Section 1 348 to 379 with setup time of 5
Job 26 Section 2 388 to 406 with setup time of 4
Job 46 Section 6 406 to 431 with setup time of 1
Maintenance 4 Section 1 431 to 467 with setup time of 0
Job 40 Section 1 467 to 498 with setup time of 1
Job 44 Section 3 498 to 503 with setup time of 4
Job 39 Section 1 503 to 511 with setup time of 5
Job 46 Section 3 511 to 516 with setup time of 1
Job 43 Section 1 516 to 531 with setup time of 2
Job 49 Section 1 531 to 539 with setup time of 5
Job 50 Section 3 539 to 551 with setup time of 1
Job 61 Section 5 551 to 571 with setup time of 4
Job 55 Section 1 578 to 613 with setup time of 5
Job 61 Section 2 613 to 621 with setup time of 4
Job 70 Section 5 630 to 646 with setup time of 5
Job 66 Section 2 646 to 651 with setup time of 3
Job 67 Section 1 651 to 669 with setup time of 3
Job 70 Section 4 669 to 684 with setup time of 5
Job 71 Section 1 697 to 734 with setup time of 5
Job 77 Section 1 734 to 747 with setup time of 5
BreakDown 7 Section 1 750 to 782 with setup time of 0
Job 96 Section 2 784 to 819 with setup time of 3
BreakDown 8 Section 1 819 to 834 with setup time of 0
Job 93 Section 3 836 to 846 with setup time of 1
Job 92 Section 1 846 to 887 with setup time of 5
Job 90 Section 1 887 to 898 with setup time of 1
Job 94 Section 2 898 to 902 with setup time of 1
Job 98 Section 1 902 to 940 with setup time of 1
Job 112 Section 2 959 to 989 with setup time of 5
Job 109 Section 1 990 to 1006 with setup time of 1
Job 111 Section 1 1006 to 1019 with setup time of 1
Maintenance 10 Section 1 1049 to 1060 with setup time of 0
Job 128 Section 1 1075 to 1117 with setup time of 5
Job 126 Section 1 1117 to 1128 with setup time of 3
Job 132 Section 1 1128 to 1134 with setup time of 3
Job 138 Section 5 1141 to 1163 with setup time of 2
BreakDown 10 Section 1 1165 to 1190 with setup time of 0
Job 136 Section 1 1190 to 1195 with setup time of 1
Job 143 Section 5 1195 to 1199 with setup time of 2
Job 145 Section 2 1199 to 1206 with setup time of 5
Job 143 Section 3 1206 to 1219 with setup time of 2
Job 157 Section 2 1280 to 1308 with setup time of 5
Job 163 Section 1 1308 to 1359 with setup time of 5
Job 164 Section 1 1373 to 1388 with setup time of 2
Job 170 Section 3 1397 to 1410 with setup time of 1
```

Schedule for Machine 4

```
Schedule for Machine 4
Job 3 Section 1 203 to 228 with setup time of 1
Maintenance 2 Section 1 236 to 262 with setup time of 0
Job 8 Section 1 262 to 268 with setup time of 5
Job 14 Section 3 268 to 277 with setup time of 1
Job 12 Section 1 277 to 299 with setup time of 5
Job 11 Section 1 301 to 315 with setup time of 4
Job 13 Section 1 316 to 324 with setup time of 3
Job 15 Section 1 324 to 328 with setup time of 2
Job 22 Section 2 328 to 341 with setup time of 3
Job 18 Section 1 341 to 368 with setup time of 4
Job 20 Section 1 368 to 380 with setup time of 2
Job 28 Section 1 380 to 397 with setup time of 2
Job 30 Section 1 397 to 418 with setup time of 1
Job 49 Section 4 418 to 424 with setup time of 5
Job 29 Section 1 424 to 437 with setup time of 1
Job 35 Section 1 439 to 451 with setup time of 1
Job 50 Section 6 452 to 455 with setup time of 1
Job 34 Section 1 455 to 481 with setup time of 5
Job 46 Section 4 481 to 483 with setup time of 1
Job 41 Section 1 483 to 505 with setup time of 4
Job 45 Section 1 505 to 528 with setup time of 4
Job 47 Section 1 528 to 541 with setup time of 5
Job 50 Section 1 541 to 551 with setup time of 1
Job 53 Section 1 552 to 571 with setup time of 5
Job 54 Section 1 571 to 576 with setup time of 4
Job 58 Section 1 591 to 618 with setup time of 5
Job 57 Section 1 618 to 621 with setup time of 2
Job 61 Section 1 621 to 626 with setup time of 4
Job 67 Section 4 639 to 649 with setup time of 3
Job 64 Section 1 649 to 675 with setup time of 3
Job 65 Section 1 675 to 680 with setup time of 1
Job 68 Section 3 680 to 684 with setup time of 2
Job 69 Section 1 686 to 705 with setup time of 1
Job 74 Section 1 705 to 727 with setup time of 2
BreakDown 6 Section 1 731 to 748 with setup time of 0
Job 79 Section 1 748 to 766 with setup time of 4
Job 78 Section 1 766 to 772 with setup time of 1
Job 80 Section 2 772 to 780 with setup time of 5
Job 81 Section 1 780 to 802 with setup time of 3
Job 85 Section 1 802 to 808 with setup time of 4
Job 94 Section 6 813 to 826 with setup time of 1
Job 86 Section 1 826 to 842 with setup time of 5
Job 94 Section 5 842 to 845 with setup time of 1
Job 88 Section 1 845 to 854 with setup time of 3
Job 91 Section 1 854 to 860 with setup time of 1
Job 89 Section 1 860 to 874 with setup time of 4
Maintenance 7 Section 1 874 to 888 with setup time of 0
Job 99 Section 1 889 to 907 with setup time of 3
Job 100 Section 1 907 to 918 with setup time of 1
Job 103 Section 1 921 to 944 with setup time of 2
Job 105 Section 1 944 to 960 with setup time of 2
Job 110 Section 1 976 to 1002 with setup time of 4
Job 116 Section 1 1003 to 1026 with setup time of 1
Job 117 Section 1 1026 to 1048 with setup time of 3
Job 119 Section 1 1055 to 1078 with setup time of 2
Job 130 Section 2 1078 to 1085 with setup time of 1
Job 132 Section 2 1085 to 1096 with setup time of 3
Job 123 Section 1 1096 to 1112 with setup time of 4
Job 129 Section 3 1112 to 1115 with setup time of 2
Job 125 Section 1 1115 to 1122 with setup time of 1
Job 130 Section 1 1122 to 1124 with setup time of 1
Job 124 Section 1 1124 to 1135 with setup time of 4
Job 133 Section 1 1135 to 1146 with setup time of 4
Job 137 Section 1 1146 to 1163 with setup time of 3
Job 138 Section 4 1163 to 1167 with setup time of 2
Job 134 Section 1 1167 to 1192 with setup time of 3
Job 142 Section 1 1192 to 1206 with setup time of 3
Job 138 Section 1 1206 to 1212 with setup time of 2
Job 143 Section 4 1212 to 1218 with setup time of 2
Job 141 Section 1 1218 to 1241 with setup time of 1
Job 144 Section 1 1241 to 1252 with setup time of 5
Job 146 Section 1 1265 to 1278 with setup time of 4
Job 147 Section 1 1278 to 1284 with setup time of 2
Job 149 Section 1 1291 to 1300 with setup time of 1
Job 154 Section 1 1306 to 1326 with setup time of 1
Job 155 Section 1 1333 to 1343 with setup time of 3
Job 162 Section 1 1349 to 1375 with setup time of 4
Job 165 Section 1 1376 to 1390 with setup time of 2
Job 169 Section 1 1402 to 1424 with setup time of 1
Job 173 Section 1 1427 to 1450 with setup time of 2
Job 176 Section 1 1455 to 1472 with setup time of 1
Job 175 Section 1 1476 to 1488 with setup time of 2
Job 178 Section 1 1507 to 1516 with setup time of 2
```

Schedule for Machine 5

```
Schedule for Machine 5
BreakDown 1 Section 1 16 to 45 with setup time of 0
Job 1 Section 1 188 to 218 with setup time of 2
Job 2 Section 1 218 to 221 with setup time of 1
Job 8 Section 5 222 to 240 with setup time of 5
Job 7 Section 1 240 to 254 with setup time of 5
Job 5 Section 1 254 to 258 with setup time of 3
Job 8 Section 2 258 to 268 with setup time of 5
Job 14 Section 4 268 to 277 with setup time of 1
Job 9 Section 1 277 to 291 with setup time of 4
Job 10 Section 1 291 to 301 with setup time of 5
Job 15 Section 2 304 to 328 with setup time of 2
Job 25 Section 2 328 to 331 with setup time of 2
Job 17 Section 1 331 to 366 with setup time of 2
Job 20 Section 2 366 to 380 with setup time of 2
Job 49 Section 5 380 to 396 with setup time of 5
Job 24 Section 1 396 to 412 with setup time of 4
Job 26 Section 1 412 to 421 with setup time of 4
Job 32 Section 1 425 to 434 with setup time of 2
Job 31 Section 1 434 to 448 with setup time of 5
Job 49 Section 2 448 to 455 with setup time of 5
Job 39 Section 2 455 to 474 with setup time of 5
Job 37 Section 1 474 to 491 with setup time of 3
Job 44 Section 4 491 to 497 with setup time of 4
Job 42 Section 1 497 to 510 with setup time of 1
Job 44 Section 1 510 to 515 with setup time of 4
Job 46 Section 1 515 to 523 with setup time of 1
Maintenance 5 Section 1 523 to 556 with setup time of 0
Job 52 Section 1 556 to 569 with setup time of 4
Job 54 Section 2 569 to 576 with setup time of 4
Job 60 Section 1 595 to 607 with setup time of 4
Job 57 Section 2 608 to 621 with setup time of 2
Job 62 Section 1 622 to 654 with setup time of 4
Job 63 Section 1 654 to 675 with setup time of 5
Job 70 Section 1 675 to 684 with setup time of 5
Job 68 Section 1 684 to 691 with setup time of 2
Job 77 Section 4 714 to 720 with setup time of 5
Job 72 Section 1 720 to 732 with setup time of 4
Job 77 Section 3 732 to 738 with setup time of 5
Job 79 Section 2 738 to 748 with setup time of 4
Job 75 Section 1 748 to 760 with setup time of 2
Job 78 Section 2 760 to 772 with setup time of 1
Job 83 Section 1 772 to 791 with setup time of 5
Job 85 Section 2 791 to 808 with setup time of 4
Job 84 Section 1 815 to 831 with setup time of 2
Job 100 Section 4 832 to 842 with setup time of 1
Job 91 Section 2 842 to 860 with setup time of 1
Job 93 Section 1 860 to 864 with setup time of 1
Job 89 Section 2 865 to 874 with setup time of 4
Job 90 Section 2 875 to 881 with setup time of 1
Maintenance 8 Section 1 881 to 905 with setup time of 0
Job 100 Section 2 905 to 907 with setup time of 1
Job 104 Section 1 914 to 927 with setup time of 5
Job 105 Section 2 931 to 944 with setup time of 2
Job 108 Section 1 947 to 979 with setup time of 1
Job 107 Section 1 979 to 1003 with setup time of 5
Job 111 Section 2 1003 to 1005 with setup time of 1
Job 114 Section 1 1005 to 1028 with setup time of 1
Job 117 Section 2 1038 to 1048 with setup time of 3
Job 122 Section 1 1048 to 1072 with setup time of 4
Job 120 Section 1 1072 to 1080 with setup time of 5
Job 130 Section 3 1080 to 1085 with setup time of 1
BreakDown 9 Section 1 1085 to 1101 with setup time of 0
Job 126 Section 3 1102 to 1110 with setup time of 3
Job 131 Section 1 1110 to 1138 with setup time of 5
Job 129 Section 1 1138 to 1142 with setup time of 2
Job 143 Section 6 1152 to 1156 with setup time of 2
Job 137 Section 2 1156 to 1163 with setup time of 3
Job 139 Section 1 1166 to 1181 with setup time of 5
Job 136 Section 2 1181 to 1190 with setup time of 1
Job 142 Section 2 1191 to 1206 with setup time of 3
Job 145 Section 1 1206 to 1227 with setup time of 5
Job 144 Section 2 1246 to 1252 with setup time of 5
Job 148 Section 1 1259 to 1275 with setup time of 1
Job 147 Section 2 1278 to 1284 with setup time of 2
Job 150 Section 1 1297 to 1321 with setup time of 2
Job 153 Section 1 1329 to 1337 with setup time of 1
Job 161 Section 1 1342 to 1358 with setup time of 1
Job 160 Section 1 1360 to 1379 with setup time of 5
Job 167 Section 1 1379 to 1386 with setup time of 2
Job 168 Section 1 1394 to 1426 with setup time of 3
Job 174 Section 1 1426 to 1446 with setup time of 1
Job 172 Section 1 1454 to 1470 with setup time of 4
```

Performance Statistics Recorded

```
JobIndex        sections        diff from deadline (Cj->dj) JobName
JobIndex 1      has 1 of max 5, ---- 0 (218->218) Job 1
JobIndex 2      has 1 of max 1, ---- 0 (45->45) BreakDown 1
JobIndex 3      has 2 of max 5, ---- 0 (221->221) Job 2
JobIndex 4      has 1 of max 9, ---- 0 (228->228) Job 3
JobIndex 5      has 1 of max 6, ---- 0 (255->255) Job 4
JobIndex 6      has 2 of max 10, ---- 0 (258->258) Job 5
JobIndex 7      has 2 of max 8, ---- 0 (288->288) Job 6
JobIndex 8      has 1 of max 6, ---- 0 (254->254) Job 7
JobIndex 9      has 5 of max 9, ---- 0 (268->268) Job 8
JobIndex 10     has 1 of max 9, ---- 0 (291->291) Job 9
JobIndex 11     has 4 of max 7, ---- 0 (301->301) Job 10
JobIndex 12     has 1 of max 8, ---- 0 (315->315) Job 11
JobIndex 13     has 1 of max 5, ---- 0 (299->299) Job 12
JobIndex 14     has 1 of max 10, ---- 0 (324->324) Job 13
JobIndex 15     has 4 of max 6, ---- 0 (297->297) Job 14
JobIndex 16     has 2 of max 7, ---- 0 (328->328) Job 15
JobIndex 17     has 1 of max 5, ---- 0 (354->354) Job 16
JobIndex 18     has 1 of max 9, ---- 0 (366->366) Job 17
JobIndex 19     has 1 of max 5, ---- 0 (368->368) Job 18
JobIndex 20     has 1 of max 6, ---- 0 (346->346) Job 19
JobIndex 21     has 3 of max 9, ---- 0 (380->380) Job 20
JobIndex 22     has 1 of max 9, ---- 0 (379->379) Job 21
JobIndex 23     has 1 of max 1, ---- 0 (184->184) Maintenance 1
JobIndex 24     has 3 of max 5, ---- 0 (372->372) Job 22
JobIndex 25     has 2 of max 7, ---- 0 (364->364) Job 23
JobIndex 26     has 1 of max 7, ---- 0 (412->412) Job 24
JobIndex 27     has 3 of max 8, ---- 0 (412->412) Job 25
JobIndex 28     has 2 of max 6, ---- 0 (421->421) Job 26
JobIndex 29     has 1 of max 9, ---- 0 (427->427) Job 27
JobIndex 30     has 3 of max 6, ---- 0 (397->397) Job 28
JobIndex 31     has 1 of max 8, ---- 0 (437->437) Job 29
JobIndex 32     has 1 of max 10, ---- 0 (418->418) Job 30
JobIndex 33     has 1 of max 10, ---- 0 (448->448) Job 31
JobIndex 34     has 1 of max 8, ---- 0 (434->434) Job 32
JobIndex 35     has 1 of max 5, ---- 0 (430->430) Job 33
JobIndex 36     has 1 of max 1, ---- 0 (262->262) Maintenance 2
JobIndex 37     has 1 of max 1, ---- 0 (253->253) BreakDown 2
JobIndex 38     has 1 of max 7, ---- 0 (481->481) Job 34
JobIndex 39     has 1 of max 8, ---- 0 (451->451) Job 35
JobIndex 40     has 1 of max 6, ---- 0 (492->492) Job 36
JobIndex 41     has 1 of max 7, ---- 0 (491->491) Job 37
JobIndex 42     has 1 of max 6, ---- 0 (476->476) Job 38
JobIndex 43     has 2 of max 8, ---- 0 (511->511) Job 39
JobIndex 44     has 1 of max 5, ---- 0 (498->498) Job 40
JobIndex 45     has 1 of max 7, ---- 0 (505->505) Job 41
JobIndex 46     has 1 of max 9, ---- 0 (510->510) Job 42
JobIndex 47     has 1 of max 9, ---- 0 (531->531) Job 43
JobIndex 48     has 5 of max 8, ---- 0 (515->515) Job 44
JobIndex 49     has 2 of max 7, ---- 0 (528->528) Job 45
JobIndex 50     has 6 of max 6, ---- 0 (523->523) Job 46
JobIndex 51     has 2 of max 8, ---- 0 (541->541) Job 47
JobIndex 52     has 1 of max 7, ---- 0 (541->541) Job 48
JobIndex 53     has 5 of max 5, ---- 0 (539->539) Job 49
JobIndex 54     has 6 of max 6, ---- 0 (551->551) Job 50
JobIndex 55     has 1 of max 1, ---- 0 (383->383) BreakDown 3
JobIndex 56     has 1 of max 6, ---- 0 (562->562) Job 51
JobIndex 57     has 2 of max 10, ---- 0 (569->569) Job 52
JobIndex 58     has 1 of max 6, ---- 0 (571->571) Job 53
JobIndex 59     has 4 of max 7, ---- 0 (576->576) Job 54
JobIndex 60     has 1 of max 8, ---- 0 (613->613) Job 55
JobIndex 61     has 1 of max 8, ---- 0 (625->625) Job 56
JobIndex 62     has 2 of max 8, ---- 0 (621->621) Job 57
JobIndex 63    has 1 of max 5, ---- 0 (618->618) Job 58
JobIndex 64    has 1 of max 6, ---- 0 (615->615) Job 59
JobIndex 65    has 1 of max 7, ---- 0 (607->607) Job 60
JobIndex 66    has 5 of max 8, ---- 0 (626->626) Job 61
JobIndex 67    has 1 of max 1, ---- 0 (457->457) Maintenance 3
JobIndex 68    has 1 of max 9, ---- 0 (654->654) Job 62
JobIndex 69    has 2 of max 10, ---- 0 (675->675) Job 63
JobIndex 70    has 1 of max 1, early 11 (467->478) Maintenance 4
JobIndex 71    has 1 of max 6, ---- 0 (675->675) Job 64
JobIndex 72    has 2 of max 8, ---- 0 (680->680) Job 65
JobIndex 73    has 2 of max 8, ---- 0 (651->651) Job 66
JobIndex 74    has 4 of max 6, ---- 0 (669->669) Job 67
JobIndex 75    has 4 of max 6, ---- 0 (691->691) Job 68
JobIndex 76    has 1 of max 5, ---- 0 (705->705) Job 69
JobIndex 77    has 5 of max 7, ---- 0 (684->684) Job 70
JobIndex 78    has 1 of max 1, ---- 0 (503->503) BreakDown 4
JobIndex 79    has 1 of max 1, ---- 0 (529->529) BreakDown 5
JobIndex 80    has 1 of max 5, ---- 0 (734->734) Job 71
JobIndex 81    has 1 of max 6, ---- 0 (732->732) Job 72
JobIndex 82    has 1 of max 10, ---- 0 (750->750) Job 73
JobIndex 83    has 1 of max 1, ---- 0 (556->556) Maintenance 5
JobIndex 84    has 2 of max 8, ---- 0 (727->727) Job 74
JobIndex 85    has 2 of max 7, ---- 0 (760->760) Job 75
JobIndex 86    has 1 of max 6, ---- 0 (734->734) Job 76
JobIndex 87    has 4 of max 10, ---- 0 (747->747) Job 77
JobIndex 88    has 2 of max 7, ---- 0 (772->772) Job 78
JobIndex 89    has 2 of max 9, ---- 0 (766->766) Job 79
JobIndex 90    has 3 of max 5, ---- 0 (786->786) Job 80
JobIndex 91    has 1 of max 9, ---- 0 (802->802) Job 81
JobIndex 92    has 2 of max 8, ---- 0 (832->832) Job 82
JobIndex 93    has 2 of max 9, ---- 0 (791->791) Job 83
JobIndex 94    has 1 of max 8, ---- 0 (831->831) Job 84
JobIndex 95    has 4 of max 10, ---- 0 (808->808) Job 85
JobIndex 96    has 1 of max 6, ---- 0 (842->842) Job 86
JobIndex 97    has 2 of max 8, ---- 0 (853->853) Job 87
JobIndex 98    has 1 of max 6, ---- 0 (854->854) Job 88
JobIndex 99    has 2 of max 6, ---- 0 (874->874) Job 89
JobIndex 100   has 1 of max 1, ---- 0 (661->661) Maintenance 6
JobIndex 101   has 2 of max 7, ---- 0 (898->898) Job 90
JobIndex 102   has 2 of max 8, ---- 0 (860->860) Job 91
JobIndex 103   has 1 of max 5, ---- 0 (887->887) Job 92
JobIndex 104   has 3 of max 7, ---- 0 (864->864) Job 93
JobIndex 105   has 6 of max 6, ---- 0 (903->903) Job 94
JobIndex 106   has 1 of max 8, ---- 0 (903->903) Job 95
JobIndex 107   has 2 of max 10, ---- 0 (916->916) Job 96
JobIndex 108   has 1 of max 7, ---- 0 (898->898) Job 97
JobIndex 109   has 1 of max 10, ---- 0 (940->940) Job 98
JobIndex 110   has 1 of max 8, ---- 0 (907->907) Job 99
JobIndex 111   has 4 of max 7, ---- 0 (918->918) Job 100
JobIndex 112   has 1 of max 6, ---- 0 (938->938) Job 101
JobIndex 113   has 1 of max 6, ---- 0 (949->949) Job 102
JobIndex 114   has 1 of max 8, ---- 0 (944->944) Job 103
JobIndex 115   has 1 of max 9, ---- 0 (927->927) Job 104
JobIndex 116   has 1 of max 1, ---- 0 (748->748) BreakDown 6
JobIndex 117   has 2 of max 7, ---- 0 (960->960) Job 105
JobIndex 118   has 1 of max 9, ---- 0 (951->951) Job 106
JobIndex 119   has 1 of max 1, ---- 0 (782->782) BreakDown 7
JobIndex 120   has 2 of max 9, ---- 0 (1003->1003) Job 107
JobIndex 121   has 1 of max 5, ---- 0 (979->979) Job 108
JobIndex 122   has 1 of max 9, ---- 0 (1006->1006) Job 109
JobIndex 123   has 1 of max 5, ---- 0 (1002->1002) Job 110
JobIndex 124   has 4 of max 7, ---- 0 (1019->1019) Job 111
JobIndex 125   has 2 of max 5, ---- 0 (989->989) Job 112
JobIndex 126   has 1 of max 8, ---- 0 (1026->1026) Job 113
JobIndex 127   has 1 of max 8, ---- 0 (1028->1028) Job 114
JobIndex 128   has 1 of max 7, ---- 0 (1039->1039) Job 115
JobIndex 129   has 1 of max 10, ---- 0 (1026->1026) Job 116
JobIndex 130   has 1 of max 1, ---- 0 (834->834) BreakDown 8
JobIndex 131   has 2 of max 7, ---- 0 (1048->1048) Job 117
JobIndex 132   has 1 of max 7, ---- 0 (1057->1057) Job 118
JobIndex 133   has 1 of max 5, ---- 0 (1078->1078) Job 119
JobIndex 134   has 2 of max 9, ---- 0 (1080->1080) Job 120
JobIndex 135   has 1 of max 6, ---- 0 (1099->1099) Job 121
JobIndex 136   has 2 of max 10, ---- 0 (1072->1072) Job 122
JobIndex 137   has 1 of max 7, ---- 0 (1112->1112) Job 123
JobIndex 138   has 1 of max 1, ---- 0 (888->888) Maintenance 7
JobIndex 139   has 1 of max 1, ---- 0 (905->905) Maintenance 8
JobIndex 140   has 1 of max 9, ---- 0 (1135->1135) Job 124
JobIndex 141   has 1 of max 7, ---- 0 (1122->1122) Job 125
JobIndex 142   has 3 of max 10, ---- 0 (1128->1128) Job 126
JobIndex 143   has 1 of max 6, ---- 0 (1139->1139) Job 127
JobIndex 144   has 1 of max 6, ---- 0 (1117->1117) Job 128
JobIndex 145   has 5 of max 5, ---- 0 (1142->1142) Job 129
JobIndex 146   has 4 of max 5, early 8 (1124->1132) Job 130
JobIndex 147   has 1 of max 5, ---- 0 (1138->1138) Job 131
JobIndex 148   has 2 of max 8, ---- 0 (1134->1134) Job 132
JobIndex 149   has 2 of max 5, ---- 0 (1146->1146) Job 133
JobIndex 150   has 1 of max 8, ---- 0 (1192->1192) Job 134
JobIndex 151   has 1 of max 7, ---- 0 (1194->1194) Job 135
JobIndex 152   has 3 of max 5, ---- 0 (1195->1195) Job 136
JobIndex 153   has 2 of max 7, ---- 0 (1163->1163) Job 137
JobIndex 154   has 5 of max 6, ---- 0 (1212->1212) Job 138
JobIndex 155   has 1 of max 5, ---- 0 (1181->1181) Job 139
JobIndex 156   has 2 of max 10, ---- 0 (1203->1203) Job 140
JobIndex 157   has 1 of max 1, ---- 0 (1007->1007) Maintenance 9
JobIndex 158   has 1 of max 7, ---- 0 (1241->1241) Job 141
JobIndex 159   has 2 of max 8, ---- 0 (1206->1206) Job 142
JobIndex 160   has 6 of max 6, ---- 0 (1219->1219) Job 143
JobIndex 161   has 2 of max 10, ---- 0 (1252->1252) Job 144
JobIndex 162   has 2 of max 9, ---- 0 (1227->1227) Job 145
JobIndex 163   has 1 of max 10, ---- 0 (1278->1278) Job 146
JobIndex 164   has 2 of max 7, ---- 0 (1284->1284) Job 147
JobIndex 165   has 1 of max 1, ---- 0 (1060->1060) Maintenance 10
JobIndex 166   has 1 of max 9, ---- 0 (1275->1275) Job 148
JobIndex 167   has 1 of max 6, ---- 0 (1300->1300) Job 149
JobIndex 168   has 1 of max 7, ---- 0 (1321->1321) Job 150
JobIndex 169   has 1 of max 5, ---- 0 (1313->1313) Job 151
JobIndex 170   has 1 of max 1, ---- 0 (1101->1101) BreakDown 9
JobIndex 171   has 1 of max 9, ---- 0 (1310->1310) Job 152
JobIndex 172   has 1 of max 9, ---- 0 (1337->1337) Job 153
JobIndex 173   has 1 of max 5, ---- 0 (1326->1326) Job 154
JobIndex 174   has 1 of max 9, ---- 0 (1343->1343) Job 155
JobIndex 175   has 2 of max 8, ---- 0 (1339->1339) Job 156
JobIndex 176   has 2 of max 6, ---- 0 (1319->1319) Job 157
JobIndex 177   has 2 of max 9, ---- 0 (1366->1366) Job 158
JobIndex 178   has 1 of max 10, ---- 0 (1362->1362) Job 159
JobIndex 179   has 1 of max 9, ---- 0 (1379->1379) Job 160
JobIndex 180   has 1 of max 6, ---- 0 (1358->1358) Job 161
JobIndex 181   has 1 of max 10, ---- 0 (1375->1375) Job 162
JobIndex 182   has 1 of max 6, ---- 0 (1359->1359) Job 163
JobIndex 183   has 1 of max 9, ---- 0 (1388->1388) Job 164
JobIndex 184   has 1 of max 8, ---- 0 (1390->1390) Job 165
JobIndex 185   has 1 of max 1, ---- 0 (1190->1190) BreakDown 10
JobIndex 186   has 1 of max 10, ---- 0 (1407->1407) Job 166
JobIndex 187   has 2 of max 5, ---- 0 (1386->1386) Job 167
JobIndex 188   has 1 of max 10, ---- 0 (1426->1426) Job 168
JobIndex 189   has 1 of max 9, ---- 0 (1424->1424) Job 169
JobIndex 190   has 3 of max 5, ---- 0 (1410->1410) Job 170
JobIndex 191   has 2 of max 7, ---- 0 (1421->1421) Job 171
JobIndex 192   has 1 of max 10, ---- 0 (1470->1470) Job 172
JobIndex 193   has 1 of max 6, ---- 0 (1450->1450) Job 173
JobIndex 194   has 2 of max 5, ---- 0 (1446->1446) Job 174
JobIndex 195   has 1 of max 1, ---- 0 (1263->1263) BreakDown 11
JobIndex 196   has 1 of max 10, ---- 0 (1488->1488) Job 175
JobIndex 197   has 1 of max 8, ---- 0 (1472->1472) Job 176
JobIndex 198   has 2 of max 8, ---- 0 (1465->1465) Job 177
JobIndex 199   has 1 of max 10, ---- 0 (1516->1516) Job 178
JobIndex 200   has 1 of max 1, ---- 0 (1287->1287) Maintenance 11
Total schedule cost = 40
Early penalty = 40 from 1 jobs
Late  penalty = 0 from 0 jobs
```

Appendix C Full Experimental Results

The full experimental results are tabulated on the following pages :

Table C1 : Heuristic Variable Settings for Tabu Search

	(1)			(2)			(3)			(4)			(5)		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T
1	229	9	0	116	3	0	305	9	0	537	15	0	17	2	0
2	10663	26	4	8746	26	2	12482	30	1	15767	30	4	9171	29	2
3	168	2	0	154	1	0	263	4	0	423	9	0	147	1	0
4	133812	32	13	130902	30	16	148396	35	14	175974	34	20	135656	30	16
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	28582	33	3	20076	38	1	36639	23	7	42795	40	6	22029	37	3
7	23710	32	3	28502	34	5	46567	23	1	45746	30	5	18983	42	1
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	460	8	0	58	4	0	730	13	0	493	12	0	176	8	0
10	35	1	0	24	2	0	167	5	0	182	5	0	0	0	0
Total	197659	143	23	188578	138	24	245549	142	32	281917	175	35	186179	149	22
Approx.	(198 k)			(189 k)			(246 k)			(282 k)			(186 k)		

Notation:

(1)	= Values used for comparison between heuristics
	TS_NEIGHBOR = 100 *ALLOWANCE* = 1.1 *ITERATIONS* = 100
(2)	= *TS_NEIGHBOR* = 300 *ALLOWANCE* = 1.1 *ITERATIONS* = 100
(3)	= *TS_NEIGHBOR* = 50 *ALLOWANCE* = 1.1 *ITERATIONS* = 100
(4)	= *TS_NEIGHBOR* = 100 *ALLOWANCE* = 1.2 *ITERATIONS* = 100
(5)	= *TS_NEIGHBOR* = 100 *ALLOWANCE* = 1.1 *ITERATIONS* = 200
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates
Pen	= Total penalty of the solution schedule

Table C2 : Heuristic Variable Settings for Genetic Algorithm

	(1)			(2)			(3)			(4)			(5)			(6)		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T
1	417	11	0	2160	22	0	1203	17	0	552	15	0	493	9	0	124	5	0
2	7663	28	0	17456	39	1	14540	44	1	9288	29	0	7306	29	0	7081	20	0
3	304	2	0	1106	11	0	912	13	0	263	4	0	186	4	0	75	3	0
4	106046	41	10	182697	29	15	162511	25	18	115982	33	14	103417	40	9	100173	41	8
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	22969	39	2	67874	31	8	51712	31	6	20284	34	3	22063	39	1	19258	35	2
7	21015	33	1	52492	34	6	50944	20	8	29048	26	7	17793	35	1	15209	33	2
8	0	0	0	235	5	0	105	4	0	0	0	0	0	0	0	0	0	0
9	820	16	0	2759	23	0	2224	16	0	717	12	0	527	10	0	496	5	0
10	156	6	0	1692	18	0	978	12	0	342	8	0	125	6	0	12	2	0
Total	159390	176	13	328471	212	30	285129	182	33	176476	161	24	151910	172	11	142428	144	12
Approx.	(159 k)			(328 k)			(285 k)			(176 k)			(152 k)			(142 k)		

Notation:

(1)	= Values used for comparison between heuristics		
		GENERATIONS = 100	*MUTATION* = 0.15 (15% mutation rate)
(2)	=	*GENERATIONS* = 100	*MUTATION* = 0 (no mutation)
(3)	=	*GENERATIONS* = 100	*MUTATION* = 0.01
(4)	=	*GENERATIONS* = 100	*MUTATION* = 0.1
(5)	=	*GENERATIONS* = 100	*MUTATION* = 0.2
(6)	=	*GENERATIONS* = 300	*MUTATION* = 0.15
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates		
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates		
Pen	= Total penalty of the solution schedule		

Table C3 : Heuristic Variable Settings for Simulated Annealing

	(1)			(2)			(3)			(4)		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T
1	2210	26	0	2717	25	0	2558	23	0	2911	24	0
2	29001	24	6	36637	30	6	26898	34	6	39489	32	6
3	1348	14	0	1679	13	0	1218	18	0	1275	15	0
4	189142	34	19	208862	27	16	176899	23	15	211014	21	16
5	0	0	0	0	0	0	0	0	0	0	0	0
6	72073	26	7	90712	33	11	77814	35	6	74681	27	6
7	86445	27	11	88870	24	9	84516	24	9	85623	24	10
8	396	8	0	426	8	0	260	6	0	286	7	0
9	3504	20	0	3703	30	0	4769	22	0	3955	26	0
10	1836	17	0	2917	15	1	1909	18	0	1854	17	0
Total	385955	196	43	436523	205	43	376841	203	36	421088	193	38
Approx.	(386 k)			(437 k)			(377 k)			(421 k)		

Notation:

(1)	= Values used for comparison between heuristics		
		START_TEMP = 1000	*TEMP_DECAY* = 0.99
(2)	=	*START_TEMP* = 700	*TEMP_DECAY* = 0.99
(3)	=	*START_TEMP* = 1500	*TEMP_DECAY* = 0.99
(4)	=	*START_TEMP* = 1000	*TEMP_DECAY* = 0.95
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates		
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates		
Pen	= Total penalty of the solution schedule		

Table C4 : Randomly spread release dates

	Greedy			Randomized Greedy			TS			SA			GA			GA + TS		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T
1	901	8	0	153	3	0	0	0	0	57	2	0	0	0	0	0	0	0
2	398	6	0	20	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1295	13	0	357	4	0	0	0	0	436	10	0	0	0	0	0	0	0
4	1848	11	0	235	5	0	0	0	0	102	4	0	0	0	0	0	0	0
5	372	7	0	25	2	0	0	0	0	0	0	0	0	0	0	0	0	0
6	367	5	0	10	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	537	8	0	20	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	452	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	398	7	0	17	2	0	0	0	0	0	0	0	0	0	0	0	0	0
10	3297	16	1	738	11	0	0	0	0	858	15	0	7	1	0	0	0	0
Total	9865	87	1	1575	30	0	0	0	0	1453	31	0	7	1	0	0	0	0
Approx.	(10 k)			(2 k)			(0 k)			(1 k)			(0 k)			(0 k)		

Notation:

Greedy	= Greedy scheduling heuristic
Randomized Greedy	= Randomized Greedy scheduling heuristic
TS	= Tabu Search heuristic
SA	= Simulated Annealing heuristic
GA	= Genetic Algorithm heuristic
GA + TS	= Genetic Algorithm followed by Tabu Search heuristic
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates
Pen	= Total penalty of the solution schedule

Table C5 : Sequential release dates

	Greedy			Randomized Greedy			TS			SA			GA			GA + TS		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T
1	247932	15	31	116914	36	13	20122	42	6	61488	41	7	13519	31	3	9174	38	1
2	595	9	0	100	4	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1841	15	0	300	9	0	0	0	0	364	8	0	0	0	0	0	0	0
4	5774	28	1	2298	21	1	0	0	0	1449	18	0	105	3	0	28	2	0
5	966	13	0	308	8	0	0	0	0	77	4	0	0	0	0	0	0	0
6	924017	22	42	486807	40	23	85095	34	12	305658	36	15	140548	54	13	75708	46	8
7	70351	22	9	2873	23	0	151	5	0	2659	27	0	321	10	0	181	5	0
8	4755	23	1	1536	22	0	0	0	0	992	14	0	30	3	0	0	0	0
9	345817	28	25	87066	46	7	20346	33	1	112336	40	7	21772	35	3	25349	37	3
10	340413	24	32	107454	31	9	2760	32	1	87959	33	7	5355	29	1	6338	30	1
Total	1942461	199	141	805656	240	53	128474	146	20	572982	221	36	181650	165	20	116778	158	13
Approx.	(1942 k)			(806 k)			(128 k)			(573 k)			(182 k)			(117 k)		

Notation:

Greedy	= Greedy scheduling heuristic
Randomized Greedy	= Randomized Greedy scheduling heuristic
TS	= Tabu Search heuristic
SA	= Simulated Annealing heuristic
GA	= Genetic Algorithm heuristic
GA + TS	= Genetic Algorithm followed by Tabu Search heuristic
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates
Pen	= Total penalty of the solution schedule

Table C6 : Release dates follow a Gaussian Curve, (-σ to σ)

	Greedy			Randomized Greedy			TS			SA			GA			GA + TS		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T
1	201	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1343	13	0	326	6	0	0	0	0	159	2	0	0	0	0	0	0	0
3	762	8	0	106	5	0	0	0	0	28	1	0	0	0	0	0	0	0
4	543	7	0	168	5	0	0	0	0	0	0	0	0	0	0	0	0	0
5	62832	23	8	13983	26	2	5306	20	0	10780	28	0	4831	23	1	4324	18	0
6	1760	13	0	1001	10	0	25	1	0	511	7	0	0	0	0	0	0	0
7	2963	21	0	1047	10	0	0	0	0	382	6	0	0	0	0	0	0	0
8	2512	17	0	1034	12	0	0	0	0	705	8	0	0	0	0	0	0	0
9	3158	20	0	648	10	0	0	0	0	359	5	0	0	0	0	0	0	0
10	1779	17	0	517	12	0	0	0	0	226	5	0	0	0	0	0	0	0
Total	77853	143	8	18830	96	2	5331	21	0	13150	62	0	4831	23	1	4324	18	0
Approx.	(78 k)			(19 k)			(5 k)			(13 k)			(5 k)			(4 k)		

Notation:

Greedy	= Greedy scheduling heuristic
Randomized Greedy	= Randomized Greedy scheduling heuristic
TS	= Tabu Search heuristic
SA	= Simulated Annealing heuristic
GA	= Genetic Algorithm heuristic
GA + TS	= Genetic Algorithm followed by Tabu Search heuristic
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates
Pen	= Total penalty of the solution schedule

Table C7 : Release dates follow a Gaussian Curve, (-2σ to 2σ)

	Greedy			Randomized Greedy			TS			SA			GA			GA + TS		
Case	Pen	E	T	Pen	E	T	Pen	E	T	Pen	E	T						
1	8641	27	1	2815	23	0	229	9	0	2210	26	0	417	11	0	237	10	0
2	154184	24	13	66181	20	12	10663	26	4	29001	24	6	7663	28	0	7656	34	0
3	3979	22	0	1161	13	0	168	2	0	1348	14	0	304	2	0	141	2	0
4	338990	28	27	255086	25	21	133812	32	13	189142	34	19	106046	41	10	104885	36	11
5	507	6	0	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	251647	15	16	133382	29	10	28582	33	3	72073	26	7	22969	39	2	19643	40	2
7	236602	14	18	117322	25	13	23710	32	3	86445	27	11	21015	33	1	22796	19	4
8	1036	13	0	600	7	0	0	0	0	396	8	0	0	0	0	0	0	0
9	17739	28	2	3691	24	0	460	8	0	3504	20	0	820	16	0	794	10	0
10	24435	17	3	4549	18	1	35	1	0	1836	17	0	156	6	0	73	2	0
Total	1037760	194	80	584792	185	57	197659	143	23	385955	196	43	159390	176	13	156225	153	17
Approx.	(1038 k)			(585 k)			(198 k)			(386 k)			(159 k)			(156 k)		

Notation:

Greedy	= Greedy scheduling heuristic
Randomized Greedy	= Randomized Greedy scheduling heuristic
TS	= Tabu Search heuristic
SA	= Simulated Annealing heuristic
GA	= Genetic Algorithm heuristic
GA + TS	= Genetic Algorithm followed by Tabu Search heuristic
E	= number of (non-maintenance non-breakdown) jobs that completed before their due dates
T	= number of (non-maintenance non-breakdown) jobs that completed after their due dates
Pen	= Total penalty of the solution schedule