CHAPTER 1

INTRODUCTION

Consider the processor of a computer as a resource tasks arrive over time needing to be processed. In which order shall these tasks be accomplished in order to minimize the average time a task is in the system?

Consider a hospital; every patient - from the doctor's perspective a task - needs a bunch of different medical treatments (like surgery, x-raying, etc.). Each treatment symbolizes a scarce resource - in most cases due to limited staff. In which order shall a given number of patients with individual needs for medical treatments be served in order to minimize the average waiting time?

Consider a certain machine within a manufacturing process as a resource, there is a set of jobs that must be processed on this machine - like semi-finished goods waiting to be completed. Each job has individual characteristics - it takes a certain time to be completed, should be finished before a certain instant of time, etc. How the jobs shall be scheduled in order to meet certain objectives?

All these scenarios have in common that a decision must be made concerning the assignment of patients, tasks, jobs to the available resources and concerning the sequence to process them on each of those resources in order to "best" fulfill a certain predetermined objective; in other words, optimal schedule must be determined.

Scheduling problems are encountered in all types of systems, since it is necessary to organize and distribute the work between many entities. A definition of scheduling problem and its components are described in different literature in different way. A definition quoted by *Carlier and Chretienne* [12]: "*Scheduling is to forecast the processing of a work by assigning resources to tasks and fixing their start times. The different components of scheduling problem are the tasks, the potential constraints, the resources and the objective function. The tasks must be programmed to optimize a specific objective function. Of course, often it will be more realistic in practice to consider several criteria*".

Ideally the objective function should consist of all costs in the system that depends on the scheduling decisions. However such costs are difficult to measure or identify completely. Hence important cost-related measures of system performance like machine idle time, job flow time, job waiting time or job lateness can be substituted for total cost and are frequently used criteria.

Another definition put forward by *Pinedo* [47]: "*Scheduling concerns the allocation of limited resources to tasks over time. It is decision – making process that has a goal the optimization of one or more objectives*".

In the above definitions, the task (or operation) is the entity to schedule. In this dissertation work we deal with jobs to schedule. When all jobs contain only a single operation we term mono operation problem. Else we say multi-operation problem. The operation of a job may be connected by precedence constraints. We deal with the resource or machine.

Effective utilization of the resources is very important in optimizing the cost, a good scheduling helps to do so. Hence, scheduling has assumed a great importance in the computer science. In the last decade there has been significant interest in scheduling practices that involve an element of batching by which there is an increase in efficiency as it may be cheaper and faster to process jobs in batches than individually. There has also been an increased research into minimizing the performance measures such as the maximum tardiness and number of tardy jobs, the former is to minimize the lateness of the latest job and the later objective is applicable when the penalty for a tardy job is the same no matter how late the job is.

Research in scheduling theory has evolved over the past forty years and has been the subject of much significant literature which uses techniques ranging from unrefined dispatching rules to highly sophisticate parallel branch and bound algorithms and bottleneck based heuristics. A diverse spectrum of researchers ranging from management scientists to production practitioners contributed to its development. However with the advent of new methodologies, such as neural networks and evolutionary computation, researchers from fields such as biology, genetics and neurophysiology, *Colorni* [14], have also become regular contributors to scheduling theory emphasizing the multidisciplinary nature of this field. Scheduling problems involve solving for the optimal schedule under various objectives, different machine environments and characteristics of the jobs. Usually, the choice of schedule has a significant impact on system performance.

An extensive literature search is done scheduling jobs on a single machine and some traditional approaches to solve like the famous Moore-Hodgson's Algorithm for minimizing the number of tardy jobs is presented. We studied a heuristic algorithm and branch and bound algorithm to solve this problem. The results obtained are compared. In this chapter introduction of machine scheduling, scheduling environment and scheduling problem are briefly discussed. The notation used and significance of the current problem dealt in this thesis is also given.

This thesis is dedicated to the problem of scheduling n jobs on a single machine. The scope is limited to deterministic problems with objective of minimizing the number of tardy jobs. A job is finished on time as long as it is completed before its due date, otherwise it is said to be tardy. Satisfying due dates is necessarily crucial, since tardiness is typically connected with extra costs.

From a theoretical point of view, scheduling problems have extensively been studied by operations research practitioners, mathematicians, production researchers and others since the beginning of the 1950s. Motivated by the numerous publications, research efforts, various

approaches and algorithms even in the field of single machine scheduling, this thesis we have addressed some deterministic problems that are tardiness related

1.1 Machine Scheduling

Everyone uses scheduling in the day-to-day life unknowingly. A student may look at the objective of completion of his class work with the constraints of assignment due dates and exams. A manager plans a schedule of operations before a project deadline against which he has to work. A manufacturer has to schedule activities in such a way as to meet the shipping dates committed to customer and also to use the available resources in an efficient manner. Sequencing and scheduling are forms of decision making which play a crucial role in both computer science, manufacturing for production scheduling and work force management for service industries. Application areas for scheduling are computer science, food, snack industries, automotive industry etc. and in service industries like workforce planning in crew scheduling, call center capacity planning, in reservation and yield management systems.

Three types of decision-making goals are prevalent, efficient utilization of resources, rapid response to demands and close conformance to prescribed deadlines. Scheduling problems are subject to two kinds of common feasibility constraints that limit on the capacity of available resources, technological restrictions on the order in which tasks can be performed. Hence scheduling deals with decision on which resource is allocated to which task and when each task is performed.

The vital elements in scheduling models are resources and tasks. They describe the type and amount of resource available, typically characterized by the qualitative and quantitative capabilities of each resource. An individual task is described in terms of its resource requirement, its duration of processing, the time at which it may be started and the time at which it is due. In addition there may sometimes existing technological constraints (precedence restrictions).

1.2 Scheduling Environments

There are many types of scheduling environments that range from small to complex. The scheduling environment may be single machine, parallel machine, classical job shop, open job shop, batch shop, flow shop or batch/flow shop.

1.3 Terminologies

1.3.1 Job a job can be made up of any number of tasks. It is easy to think of a job as making a product and each task as an activity that contributes to making that product, such as a painting task, assembling task and so on.

1.3.2 Machine, A machine is available to execute jobs and tasks. Different machine environments exist, such as single machine and parallel machines.

1.3.3 Processing time, Length of time required for processing a job or a task.

1.3.4 Release Time, Time at which job is released for processing.

1.3.5 Completion time, Time at which processing of a job is finished.

1.3.6 Due date, The latest time by which a job should be completed so that it is not late.

1.3.7 Lateness It is a measure of how late a job is completed. The difference between completion time and the due date of a job give the lateness. $L_j=C_j - D_j$, where C_j is the completion of job j and D_j is the due date of job j.

1.3.8 Tardiness, The tardiness of job j, T_j, is defined as $T_j = \max \{0, C_j-D_j\}$

1.3.9 Preemption, The Preemption (or job-splitting) is allowed during the processing of a job, if the processing of the job can be interrupted at any time (preempt) and resumed at a later time, even on a different machine. The amount of processing already done on the preempted job is not lost.

1.3.10 Setup, Many practical scheduling problems involve intermediate delays between processing of successive jobs or between the operations of same job. It may reflect the need to change the tools or to clean a machine. This is the unproductive time on the machine but is necessary when there is a switch from one family type to another.

1.3.11 Batch, It is execution of a series of programs ("jobs") on a computer without manual intervention. In some manufacturing environments, it's more efficient to process a subset of jobs (called a batch of jobs) simultaneously.

1.4 Scheduling Problem

Scheduling problems are concerned with the allocation of resources over time to perform tasks. Tasks can be seen as jobs and the resources are typically divided into processors (machines) and additional resources. Thus, scheduling means allocating scarce production resources (machines) in order to complete tasks (jobs) under certain constraints with the aim to best fulfill specific

objectives. The goal is the determination of a schedule that specifies when and on which machine each job is to be executed.

1.5 Three-field (α|β|γ) Notation

It is popularly used to denote the scheduling problems defined by Lawler [39]. It consists of three fields. A problem is denoted by **α|β|γ,** where the field α describes the machine environment. Field β describes the jobs and their interrelations, and field γ denotes the objective function. Latter, it is described in Chapter 3. Our problem of scheduling jobs on a single machine to minimize the number of tardy jobs with release time constraints is denoted by $1|\ r_j\ |\sum U_j$.

1.6 Notation Used in this Thesis

J = {1, 2,…, n}, job set to be processed

$\mathbf{p_j}$ processing time of component, $1 \leq j \leq n$

$\mathbf{r_j}$ release time of the component, $1 \leq j \leq n$

$\mathbf{d_j}$ due date of job j, $1 \leq j \leq n$

$\mathbf{T_j}$ tardiness of job j, $1 \leq j \leq n$, $T_j = \max\{0, C_j - D_j\}$

$\mathbf{U_j}$ status flag of job j, $1 \leq j \leq n$;

$$\mathbf{U_j} = \begin{cases} 1, & if\ job\ j\ is\ tardy \\ 0, & otherwise \end{cases}$$

$\mathbf{C_j}$ completion time of job j, $1 \leq j \leq n$

1.7 Significance of the Problem

Problem considered in this research deals with scheduling n jobs on single machine to minimize the number of tardy jobs with release time constraints, each job to be completed have different release dates, processing time and the due date. This is proved to be NP-Hard problem and is only solvable in pseudo-polynomial time and has very much importance in computer science.

1.8 Organization of the Thesis

The Chapter 1 briefly describes the introduction to scheduling, significance of the problem, machine scheduling and scheduling problem and also introduces the scheduling environment.

The Chapter 2 describes algorithms and computational complexity. The theoretical basis of computer science has been formulated. Computational resources and complexity classes are described.

The Chapter 3 describes the formal description of scheduling theory, different types of scheduling problems and solution strategies.

The Chapter 4 discusses the single machine scheduling problems and methods previously used to solve similar scheduling problems in the literature.

The Chapter 5 describes the past work in the similar problem and some traditional approach to solve the problem.

The Chapter 6 describes heuristic algorithm and branch and bound algorithms to solve the problem.

In the Chapter 7 computational experiments are performed on randomly generated real size data and the solutions obtained for both heuristic algorithm and B&B are compared.

In Chapter 8 Conclusion and directions of future research is given.

CHAPTER 2

COMPUTATIONAL COMPLEXITY

Computational complexity analysis, is branch of the theory of computation in computer science, investigates the problems related to the amounts of resources required for the execution of algorithms (e.g., execution time), and the inherent difficulty in providing efficient algorithms for specific computational problems. In particular, the theory places practical limits on what computers can accomplish. *Blum* [8] developed axiomatic approach to measure computational complexity**.** He introduced such measures of complexity as the size of a machine and axiomatic complexity measure of recursive functions. The time complexity and space complexity of algorithms are special cases of the axiomatic complexity measure; the axiomatic approach helps to study computational complexity in the most general setting. An important aspect of the theory is to categorize computational problems and algorithms into complexity classes.

2.1 Turing Machines and Algorithms

It all started with a machine. In 1936, Alan Turing developed his theoretical computational model. His model is based on his perception of the way mathematicians think. As digital computers were developed in the 40's and 50's, the Turing machine proved itself as the right theoretical model for computation. Basically, Turing machine converts one set of strings to another. A Turing machine comprises of a finite control, a tape, and a head that can be used for reading and writing on the tape.

There is no precise definition of an algorithm. It is believed that algorithms and Turing machines are equivalent. That is every problem solvable by Turing machine is solvable by algorithm and vice versa. There are other models of computations apart from Turing machines and algorithms .Church Turing's thesis states that all these models are equivalent, *Deutsch* [28]**.** An algorithm is any well defined computational procedure that takes some value or set of values as input and produces some value or set of values as output.

The basic Turing machine model fails to account for the amount of time or memory needed by a computer, as critical issue today but even more so in those early days of computing. The key idea

to measure time and space as a function of the length of the input came in early 1960' by *Hartmanis and Stearns* [25] and thus computational complexity was born.

2.2Computational Resources

Complexity theory analyzes the difficulty of computational problems in terms of many different computational resources. The same problem can be explained in terms of the necessary amounts of many different computational resources, including time, space, randomness, and other less-intuitive measures. A complexity class is the set of all of the computational resource.

The most well-studied computational resources are time and space. The time complexity of a problem is the number of steps that an algorithm takes to solve an instance of the problem. The space complexity of a problem measures the amount of space, or memory required by the algorithm. A good algorithm always takes less time and less space. A better algorithm in bad machine may appear insufficient compared to bad algorithm in good machine. To minimize effect of these considerations, computational complexity deals with instances whose input size is very large, so that machine size can be neglected. To describe behavior of algorithm for large input the concept of asymptotic order is useful.

2.3 Functions

Given two sets A and B, a function f is a binary relation on A×B such that for all a∈A, there exits precisely one b∈B such that (a, b)∈f. The set A is called domain of f, and the set B is called co-domain of f. we write f: A→B and if (a, b) ∈ f, we write b=f (a), since b is uniquely determined by choice of a. Two functions f and g are equal if they have the same domain and co-domain and if, for all a in the domain, f (a) =g (a).

A finite sequence of length n is function f whose domain is the set of n integers {0, 1, 2,…, n-1}. Finite sequence is denoted by listing its values :{ f (0), f (1), f (2),…,f (n-1)}. An infinite sequence is a function whose domain is set of N natural numbers. For example, the Fibonacci sequence, defined by recurrence, is the infinite sequence {0, 1, 1, 2, 3, 5, 8,…}.

A function f(x) between two ordered set is unimodal if for some value m (the mode), it is monotonically increasing for $x \leq m$ and monotonically decreasing for $x \geq m$. In that case, the maximum value of f(x) is f (m) and there are no other local maxima. Examples of unimodal function: quadratic polynomial, logistic map, tent map.

A function is convex if and only if its epigraph (the set of points lying on or above the graph) is convex set. Pictorially, a function is called 'convex' if the function lies below the straight line

segment connecting two points, for any two points in the interval. A function f is said to be concave if -f is convex.

2.3.1Asymptotic Order of Functions

There are three asymptotic orders that are commonly used. They are big-O, big-Omega and big-Theta, Let f: N→ R^+ and g: N →R^+ be two functions from the set of natural numbers to the set of non-negative real numbers.

The function f (n) =O (g (n)) (reads as "f of n is big oh of g of n") if and only if there exist positive constants c and n_0 such that f (n) ≤c*g (n) for all n such that n≥n_0.

The function f (n) =Ω (g (n)) (read as "f of n is omega of g of n") if and only if there exists positive constants c and n_0 such that f (n) ≥c*g (n) for all n such that n≥n_0.

The function f(n)= Θ (g(n))(read as "f of n is theta of g of n") if and only if there exists positive constants c_1,c_2 and n_0 such that c_1*g(n)≤f(n)≥c_2*g(n) for all n such that n>n_0.

2.4Time and Space Complexities of Algorithms

Time requirement is counted in units of steps. Space requirement is counted in units of memory cells. For any algorithm, one may have not specified time or space complexity or both for example, if an algorithm has time complexity of O (f (n), then it means that the number of steps required by the algorithm is bounded above by f (n). Space complexity can be stated similarly.

Usually in computational complexity theory, one considers time complexity. In the following discussions; the term 'complexity' is used to denote time complexity unless explicitly mentioned.

2.5Problems and Encoding

A computational problem can be viewed as a function f which maps each input x in some domain to an output f(x) in some given range. An instance is a member of the input domain. Computational problems are called abstract problems whenever their peculiarities are unknown. However, for formal treatment, one has to restrict attention to typical problems.

Here some important types of problems in computational complexity theory are mentioned, a problem whose output can either be 'yes' or 'no'. Scheduling problems belong to the class of discrete optimization problems. For this, the following definitions of an instance of optimization problem and an optimization problem are needed: An instance of an optimization problem is a pair (F, c), where F is any set, the domain of feasible points, c is the cost function, a mapping $c:F \rightarrow R^+$. The problem is to find $f \in F$ such that for all g, f c (f) $\leq$c (g) (some optimization problem may use $\geq$ instead of $\leq$ such problems are called maximization problems). An optimization problem is a set I of instances of an optimization problem.

Conceptually, a particular instance of an optimization problem can have many feasible solutions: F is the set of such solutions. By feasible solutions, solutions that do not violate the given constraints are understood. Among the feasible solutions from F, the solution having minimum cost is to be selected. This cost is given by cost function c: $F \rightarrow R^+$.

Now the concept of discrete optimization problems is defined: an optimization problem whose all instances have finite set of feasible solutions is known as a discrete optimization problem. Discrete optimization problems are also known as combinatorial optimization problems. Large classes of combinatorial optimization problems are important tools to solve problems in computer science and interpretation technology.

To be computed by an algorithm, problem instances should be represented in a suitable way. Generally, binary strings are used to this purpose. First consider the definition of an encoding: An encoding of a set S of abstract objects is a mapping e from S to the set of binary strings. Generally the set {0, 1} is used as an alphabet for binary encoding. Let e is an encoding of abstract objects. Let $s \in S$, then the length of encoding denoted by |e(s)|, is the number of symbols in e(s).

There is a special reason for choosing binary string instead of unary or others .Actually the efficiency of an algorithm is affected by the encoding used. Time and space complexities are described in terms of length of the encoded string. This shows that the encoding scheme can not be let arbitrary. However, binary encoding has an advantage over unary. Because the length of an input string in a higher order encoding differs from length in binary encoding by a polynomial factor only. This can be seen by changing base of logarithm while converting a string in higher order encoding to binary encoding, the complexity remains same. The only trouble is with unary encoding.

2.6 Complexity Classes

A complexity class is the set of all of the computational problems which can be solved by an abstract machine M using O (f (n)) of resource R, where n is the size of the input. There are several complexity classes in the theory of computation. The major classes are discussed below.

2.6.1 Classes P and NPs

The P class comprises all decision problems that can be solved with polynomial computational effort. This class corresponds to an intuitive idea of the problems which can be effectively solved in the worst cases.

Example2.1 The problem of sorting n numbers can be done in O (n^2) time using the quick sort algorithm in worst case. Thus all sorting problems are in P.

NP stands for nondeterministic polynomial time. The class comprises all problems for which a certificate can be checked by an algorithm with polynomial computational effort. This class contains many problems that people would like to be able to solve effectively, including the Boolean satisfiability problem, the Hamiltonian path problem and the vertex cover problem. All the problems in this class have the property that their solutions can be checked efficiently.

Example 2.2 A vertex cover of an undirected graph G= (V, E) is a subset of V' $\subseteq$V such that if (u, v) $\in$ E, then u $\in$ V' and v $\in$ V' or both. That is, each edge touches at least one vertex V'. The vertex –cover problem is to find such a vertex cover of minimal cardinality. This problem is in NP Class.

The P=NP Question

The question of whether NP is the same set as P (that is whether problems that can be solved in non-deterministic polynomial time can be solved in deterministic polynomial time) is one of the most important open questions in theoretical computer science due to the wide implications a solution would be present. If it were true, many important problems would be shown to have "efficient" solutions. These include various types of integer programming in operations research, many problems in logistics, protein structure prediction in biology, and the ability to find formal proofs of pure mathematics theorems efficiently using computers. The P=NP problem is one of the Millennium prize problems proposed by the Clay Mathematics Institute the solutions.

2.6.2 Classes NP-Complete and NP-Hard

NP-Complete

NP-complete are the hardest problems among the NP class. The class NP-complete is the set of decision problems X such that
1. $X \in NP$.
2. Every problem in NP is reducible to X .i.e. NP–complete are the hardest problems among the NP-class.

NP-Hard

NP-Hard can contain problems other than decision problems, it is a class of all problems X such that for all $Y \in NP$, $Y \leq_P X$, That is there may be a problem X which is as hard as any problem in NP, but one may not be able to prove its NP completeness.

Other can be solved with polynomial effort as well. When expanding the view by including optimization problems, one distinguishes between two classes.

Regardless the fact that all NP-hard /NP-complete problems are computationally hard, some of them may still have tolerably efficient exact algorithms. Such algorithms, known as pseudo-polynomial algorithms have complexities slightly higher than polynomials, on the other hand, there are strongly NP-hard and strongly NP-complete problems: under the assumption $P \neq NP$, strongly NP-hard and NP-Complete problems can not have pseudo-polynomial algorithms .For example, the problem TSP (Travelling Salesman Problem) is strongly NP-Hard, thus two NP-hard or NP-complete problems may not be computationally equivalent. For detail we refer [46]. Due to these reasons, scheduling problems have also been classified according to the hierarchy of complexity.

.

CHAPTER 3

SCHEDULING PROBLEMS

In this chapter, the basic formulations of the scheduling theory are described. Since the domain of scheduling theory is very wide, only single machine scheduling problems are considered as far as possible.

3.1Representation of Schedule

Let there be m number of machines, M_i, i=1,2,..,m which have to process n jobs, J_i, i=1,2,…,n. The problem is to assign each job one or more time intervals on one or more machines. Such an assignment is called a schedule in general term. A schedule is often represented by Gantt chart. Below is an example of schedule for a single machine.

The schedule of jobs can be represented as a sequence of jobs. For example, the schedule shown in Figure: 3.1 can be written as the sequence s= (J_1, J_4, J_2, J_3). The machine may remain idle for some time interval. We specify idle intervals by writing 'idle' for that time interval.

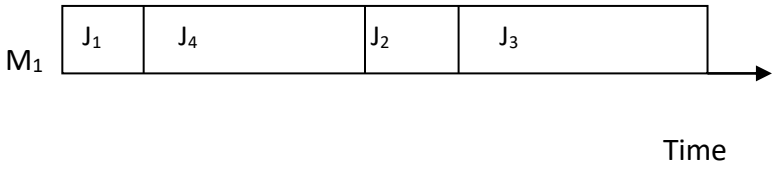

Figure 3.1: Gantt chart for schedule of four jobs in single machine

3.2Machine Environment

There can be a single machine, multiple machines or in some situations, the number of machines may be unknown in advance. Let us briefly discuss the multiple machine environments, which is the general case. On the multiple machine environments, a job J_i is a set of n_i numbers of operations O_i. It is not necessary that an arbitrary operation of an arbitrary job

can be processed in an arbitrary machine: this restriction inspires to classify the multiple machine environments into two groups; parallel machines and dedicated machines.

In parallel machine model, an arbitrary operation O_{ij} of an arbitrary job J_i can be executed in an arbitrary machine M_j Stating simply, any machine can execute any operation of any job.

In the dedicated machine model, there is a restriction on operations, operations executable on machines are constrained. To be specific, dedicated machine environment has been classified into three categories,viz.., flow shop, open shop and job shop.

Consider a job j_i with n_i operations, O_{1i},O_{2i},…O_{ni}. In an open shop, the number of operations is same for all jobs, say m. Further, the operation o_{1i} should be processed on M_1, O_{2i} on M_2 and in general, O_{ki} on M_k.

3.3Job Description

Each job J_j is provided with a number p_j, which is the processing time of J_j. This definition of processing time is sufficient for single machine environment. For the multiple machine environments processing time of each operation of each job may differ from machine to machine, so the processing times are often provided in matrix form. Here, only the single machine case is considered.

Another important data regarding a job J_j is its release date, r_j. Release date means the time by which the job is ready for processing. Similarly, for each job J_j, there may be weight w_j, due date d_j, and a deadline d_j, weight of a job means its priority.

There can be precedence relation among the job. In general, a precedence relation is a directed acyclic graph (DAG), G (V, E), where the vertices V are the jobs, and $(J_i,J_j)\in E$ if and only if J_i precedes J_j, In specific situations, precedence relation can occur as a tree, or some other special type of DAG like sp-graphs. A tree can be outtree or an intree. In outtree, the DAG is a rooted tree with indegree for each vertex at most one. Similarly, an intree is a rooted DAG with outdegree for each vertex at most once.

3.4 Characteristics of a Good Schedule

In scheduling problems, finding a good objective to minimize or maximize can be difficult and it may be crucial step for several reasons. Firstly, important objectives like customer satisfaction for quality or promptness are difficult to quantify and do not appear among the accounting numbers, these enhance the quality of service. A shop usually deals with three types of

objectives, such as Maximize shop throughput over some time period, satisfy customer desires for quality and promptness and minimize costs.

A schedule that is developed considering one of the above three objective functions can be considered a good schedule. Several approaches can be considered to achieve one or more of these objectives

1. To solve problems with one objective at a time
2. To solve for trade-of curves between objectives and
3. To combine objectives by assigning costs to customer desires and lack of utilization.

Scheduling can be difficult from both a technical and implementation point of view.

There are three characteristics which cause scheduling to be extremely difficult to automate.

1. Its combinatorial complexity (scheduling is an NP-Hard problem);
2. The conflicting nature of the requirements of a "good" schedule;
3. The uncertainty of execution caused by the stochastic and dynamic nature of most scheduling environments.

The most prominent feature of scheduling problems is their combinatorial complexity. Most combinatorial optimization problems, except a few very simplistic ones are in the NP class, and are therefore difficult to solve. A second difficulty with scheduling involves assessing the value of a scheduling decision within a specific domain. Often it is unclear how one decision will influence the global satisfaction of organizational goals such as machine utilization, due date satisfaction or work in progress levels.

The need to optimize both the quantitative as well as qualitative objectives such as minimization of costs along with customer satisfaction with quality or promptness makes the problem more complex. Thirdly, unforeseen events like machine breakdown, power failure, and personnel absences in a machine shop may invalidate a schedule. An effective schedule should be able to adapt to these environmental and executional uncertainties. Hence it is very complex problem far from being completely solved to optimality in most of the cases. In fact, it is computationally impossible to check for optimality for most scheduling problems.

3.5 Three-field (α|β|γ) Notation

This notation is given in *Lawler et al* [39]. It is popularly used to denote the scheduling problems. It consists of three fields. A problem is denoted by **α|β|γ,** where the field α describes

the machine environment. Field β describes the jobs and their interrelations, and field γ denotes the objective function.
The field, in general, is $\alpha = \alpha_1 \bullet \alpha_2$ Where • denotes string concatenation.

Parameters $\alpha_1 \in \{\emptyset, P, Q, R, O, F, J\}$ characterizes the type of machine used. Specifically,

$\alpha_1 = \emptyset$: single machine ($\emptyset$ denotes empty string)
α_1=P: identical machines.
α_1=Q: uniform machines.
α1=R: unrelated machines
α_1=O: dedicated machines, open shop system.
α_1=F: dedicated machines, flow shop system.
α_1=J: dedicated machines, job shop system.

Parameter $\alpha_2 \in \{\emptyset, k\}$ denotes the number of machines.
$\alpha_2 = \emptyset$: number of machines is assumed to be variable.
α_2=k: k number of machines.

The β –field, in general, is $\beta = \beta_1 \bullet \beta_2 \bullet \beta_3 \bullet \beta_4 \bullet \beta_5 \bullet \beta_6$
Parameter $\beta_1 \in \{\emptyset$,pmtn$\}$ denotes whether preemption is allowed or not.
$\beta_1 = \emptyset$:preemption is not allowed.
β_1=pmtn: preemption is allowed, i.e., a job being processed can be paused arbitrarily, and start some another available job.

Parameter $\beta_2 = \emptyset$: no resource constraints.
β_1=res: resource constraints are given.
Parameter $\beta_3 \in \{\emptyset$,rec,tree$\}$ denotes precedence constraints.
$\beta_3 = \emptyset$: no precedence constraint.
β=prec:precedence is given in the form of an arbitrary DAG.
β_3=tree: precedence is given in the form of a tree.
β_3=intree: precedence is given as an intree.
β_3=outtree: precedence is given as an outtree.

Parameter $\beta_4 \in \{\emptyset, r_j\}$ describes release dates.
$\beta_4 = \emptyset$: release date is zero for all jobs (or equal release dates)
$\beta_4 = r_j$: release date is given for each job.
Parameter $\beta_5 \in \{\emptyset, p_j = p\}$ describes the processing times.
$\beta_5 = \emptyset$: jobs have arbitrary processing times.
$\beta_5 = (p_j = p)$: all jobs processing times equal to p.
Parameter $\beta_6 \in \emptyset$, d} denotes whether deadlines are given or not.
$\beta_6 = \emptyset$: no deadlines.
β_6=d: jobs have deadlines.

In the **γ**-field, the formula or a short symbol for denoting the objective function is simply written. For example, one can write $\sum_j w_j C_j$ to indicate weighted completion time has to be minimized. Here are some examples of the three-field notation $1|r_j|\sum_j w_j C_j$ denote the single machine problem where release dates are given, and the objective is to minimize the weighted completion time. $1||F_{var}$ denotes the single machine where the flow time variance has to be minimized. $P|prec, p_j=1|C_{max}$ denotes identical parallel machines, precedence relation is given as DAG, all jobs have unit processing time, and the objective is to minimize the maximum completion time.

3.6 Scheduling in Computer Science

Scheduling is a fundamental operating–system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating–system design. Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short–term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
For detail we refer [44].

Scheduling problems posed by Operating Systems (OS) are online versions of various scheduling problems. In an online version, one does not know processing time and other relevant information of a job until it actually arrives in the system. In an OS, a machine is a processor, and jobs are processes (a process is a program ready for execution). The machine environment has a vast variety. There can be multiple processors, preemption may or may not be allowed, and in almost all situations, the scheduling problems are resource constrained. OS designers take engineering approach due to this variation. The scheduling algorithms are selected on the basis of simulation experiments. Objective function for OS oriented scheduling is different than those for manufacturing companies. A manufacturing company aims to reduce production cost, where as an OS aims to provide a fair service to all user processes. This leads objective functions like:

1. Processor utilization: This is the average function of time during which the processor is busy.
2. Throughput: This is the number of processes executed per unit time. Throughput is computed by dividing number of processes by schedule length.
3. Average turnaround time: The time that elapses from the moment a program released until it is completed by the system.

4. Average waiting time: The time that a process spends waiting for the processor or some other resources.

5. Average response time: The time taken by a process to response after it is released.

Scheduling problems in computer system is mostly based on the analysis of queuing theory. The basic queuing model is given below.

Jobs arrive and wait in a queue. The queue is the main memory for an OS. Every scheduling algorithms of an OS follows this model. Some basic algorithms used in OS for uni-processor computers are given below.

1. First come First Serve (FCFS): At any instant when machine is idle, select available job having least release date.
2. Shortest processing Time(SPT): When the machine is idle select the available job having least processing time, This rule is also called Shortest Job First(SJF).
3. Shortest Remaining Time Next (SRTN): Select an unfinished job which is having the smallest remaining processing time.
4. Rounds Robin: Available jobs are stored in a queue according to release dates, unit processing time is given to each job in a queue in the sorted order. The newly arrived job is appended to the queue; Completed jobs are removed from the queue.

3.7Online Problems

The discussion so far is considered with offline version of scheduling. In an online version, one does not know processing time and other relevant information of a job until it actually arrives in the system. This online model is realistic. For example, consider the online version of the problem $1|r_j|\sum C_j$. This problem has a deep significance in operating system design. One of the goals that the operating system seeks to achieve is to minimize the average response time, i.e. ,$(1/n)\sum(s_j-r_j)$, where s_j is the starting time of a job J_j, since $s_j=C_j-p_j$, and p_j is constant for all job j, this problem is equivalent to minimizing total completion time. Not only scheduling but many computational problems in general have online versions.

In the field of scheduling, as well as many related topics, the online versions are getting more and more attention. Many of the classical problems recur in modern online applications, but with slightly modified objective functions, among these new objective functions, flow time and stretch are important ones. Flow time was defined in applications like handling requests at web servers, scheduling jobs in operating systems and parallel computing scheduling problems in which flow time and /or stretch has to be minimized. The latest research on online algorithms is

described by *Jawor* [29]. Unless specifically mentioned, the discussion throughout this document considers offline version of scheduling problems.

An interesting feature of online algorithms is that despite their complexity, commonly used algorithms for solving them are very simple and easy to implement. In an online environment, scheduling decision has to be taken in a very short interval of time and such decisions have to be taken again and again. So, one does not have the time to find exact solution, simple algorithms giving reasonable solutions are acceptable.

3.8Just –in-Time and Real-Time Scheduling

Just-in-Time is a management system of producing only what is required, in the exact quantity required, exactly when it is required and delivered exactly where it is required on time. Take an example of a manufacturing company. If items are produced earlier that the expected date, then they have to be preserved, and thus add unwanted storage costs. On the other hand, late production can lead to fines, express delivery charges, lost sales, etc. A just-in-time schedule minimizes sum of earliness and tardiness penalties.

A special type of just in time scheduling is due date scheduling. Basically; there are two versions of this problem. In the first version, a common due date is given for all jobs; one has to find schedule minimizing lateness and tardiness penalties with respect to this due date. The second version is reverse of the first one, here, one has to determine a common due date such that the penalties are minimized. The symbols'd' and 'd_{opt}' are added in the β field of the three field notation to indicate first and second versions due date scheduling, respectively. Unlike other scheduling problems, usually due date scheduling considers positional weights. This means, weight w_j does not correspond to job J_j, but to any job that occurs in position j of the schedule. For detail we refer ([20], [21]).

Real-time scheduling problems are principally online versions of just-in-time scheduling problems, but popularly, the nomenclature 'real-time' refers to computer related problems. These types of scheduling problems occur in real –time systems. Generally a real –time system is an operating system embedded in some electronic devices. In a real-time system, the correct functioning of the system depends on the time when jobs are completed. In a soft real-time system, early/tardy jobs degrade the quality of the output, while in a hard real-time system; such jobs make the output invalid. The book of *Tanenbaum* [50] provides an introduction for real-time scheduling problems in operating systems.

3.9 Other Problems

We encounter scheduling problems in computer systems. These problems are studied in different forms by considering mono or multi processor systems, with the constraints of synchronization of operations and resource sharing. In these problems, certain operations are periodic others are not; some are subject to dates, others to deadlines. The objective is to find a feasible solution i.e. a solution which satisfied the constraints. In fact, in spite of appearances they are very close to those encountered in manufacturing systems, *Blazewicz et al* [7].

Timetable scheduling problems concern all educational establishments or universities, since they involve timetabling of courses assuring the availability of teachers, students and classrooms. These problems are just as much the object of studies.

Project scheduling problems comprise a vast literature. We are interested more generally in problems of scheduling operations which use several resources simultaneously (money, personnel, equipment, raw materials etc.), these resources being available in known amounts. In other words, we deal with the multi-resource scheduling problem with cumulative and non-renewable resources.

3.10 Types of Scheduling Problems

Scheduling problems can be classified in terms of number of machines, flow discipline, job availability (in case of batching), and so on.

3.10.1 Single Machine

The single machine environment is very simple and basic job shop problem consisting of a single processor and n jobs. Each job has one operation to be performed. It provides a basis for heuristics for more complicated machine environments. Complicated machine environments are often decomposed into single machine sub problems. Polynomial time or heuristic algorithms are already found for most of the single machine cases. For single machine problems it has been proved that SPT is optimal for finding the minimum total completion time, weighted completion time, average flow time and EDD gave better schedules for performance measures like maximum lateness, number of tardy jobs, total tardiness and other due date related objectives.

3.10.2 Parallel Machines

Multiple machines are available to process jobs. The machines can be identical, of different speeds, or specialized to only processing specific jobs. Each job has a single task.

3.10.3 Flow Shop

In the general flow shop model, there are a series of machines numbered 1, 2, 3 … m. Each job has exactly m tasks. The first task of every job is done on machine 1, second task on machine 2 and so on. Every job goes through all m machines in a unidirectional order. However, the processing time each task spends on a machine varies depending on the job that the task belongs to. In cases where not every job has m tasks, the processing times of the tasks that don't exist are zero. The precedence constraint in this model requires that for each job, task i-1 on machine i-1 must be completed before the i^{th} task can begin on machine i.

3.10.4 Job Shop

In the general job shop model, there are a set of machines indexed by k, jobs indexed by i, and tasks indexed by j. Each task on a machine is indicated by a set of three indices i, the job that the task belongs to, j, the number of the task itself, and k, the machine that this particular task needs to use. The flow of the tasks in a job does not have to be unidirectional. Each job may also use a machine more than once.

1.10.5 Static

In this problem, the activities are known in advance and the constraints are fixed. It assumes all job arrivals at time zero (i.e., all the raw material for all the jobs is available at the beginning) and the resources are always available without any breakdowns throughout the production process.

3.10.7 Dynamic

This problem is more realistic and difficult to solve as compared to the static case. Every scheduling problem is subject to unexpected events. The job arrivals are not static and can be released into the system randomly. The resources may be unavailable due to unforeseen breakdowns at anytime. In these cases, a new solution is needed in a preferably short time taking these events into account and as close as possible to the current solution.

3.11 Solution Methodologies

A variety of techniques are used for determining solutions to scheduling problems. Dynamic programming, network methods, heuristic solution approaches, dispatching rules and simulation techniques are used depending on the complexity of the problem, nature of the model, and the choice of the criterion. Hence scheduling deals both with the study of models as well as the methodologies used to solve the problem. Dispatching (or priority) rules and heuristic search techniques do not guarantee finding an optimal solution, but aim at finding reasonably good solutions in a relatively short time, while the dynamic programming or branch and bound techniques search a large solution space and hence take a long time to find a solution.

3.11.1 Dispatching Rules

The jobs are arranged in a list according to some rule. The next job on the list is then assigned to the first available machine. The following are some of the common rules.

3.11.2 Random List

This list is made according to a random permutation.

3.11.3 Longest Processing Time (LPT)

The longest processing time rule orders the jobs in the order of decreasing processing times. Whenever a machine is freed, the longest job ready at the time will begin processing. This algorithm is a heuristic used for finding the minimum make span of a schedule with parallel machines. It schedules the longest jobs first so that no one large job will "stick out" at the end of the schedule and dramatically lengthen the completion time of the last job.

3.11.4 Shortest Processing Time (SPT)

The shortest processing time rule orders the jobs in the order of increasing processing times. Whenever a machine is freed, the shortest job ready at the time will begin processing. This algorithm is optimal for finding the minimum total completion time and weighted completion time, if there is a single machine. In the single machine environment with ready time at 0 for all jobs, this algorithm is also optimal in minimizing the mean flow time, minimizing the mean number of jobs in the system, minimizing the mean waiting time of the jobs from the time of arrival to the start of processing, minimizing the maximum waiting time and the mean lateness.

3.11.5 Earliest Due Date (EDD)

In the single machine environment with ready time set at 0 for all jobs, the earliest due date rule orders the sequence of jobs to be done from the job with the earliest due date to the job with the latest due date. Let D_i denote the duedate of the i^{th} job in the ordered sequence. EDD sequences jobs such that the following inequality holds $D_1 \leq D_2 \leq \ldots \leq D_n$. EDD finds the optimal

schedule when one wants to minimize the maximum lateness, or to minimize the maximum tardiness.

3.11.6 Dynamic Programming Approach

It is basically a complete enumeration scheme that minimizes the amount of computation to be done by dividing the problem into a series of sub-problems. It solves these sub-problems until it finds the solution of the original problem. It determines the optimal solution of the sub-problem along with its contribution to the objective function and tries to improve the solution by a number of iterations, boundary conditions; a recursive relation and an optimal value function characterize it. It is a reasonable approach to find an optimum sequence for problems where efficient optimizing procedures have not been developed. It is typical in many general-purpose procedures for combinatorial optimization. Though the effort required to solve the problem grows exponentially with the problem size, it is more efficient than complete enumeration as computational demands for the latter increases as a factorial of the problem size. It is often described as implicit enumeration as it considers certain sequences only indirectly, without actually evaluating them. In scheduling, the forward dynamic programming and backward dynamic programming approaches can be used.

3.11.7 Branch and Bound Approach

It is a general-purpose strategy for curtailed enumeration. As the name implies, it branches i.e., a large problem is partitioned into two or more sub-problems and each of these small problems are solved to obtain best solutions by calculating a lower / upper bound on the optimal solution of a given sub-problem. Many branches of sub problems are eliminated by dominance properties and a list of unsolved sub-problems that have been encountered are sufficient to determine an optimal solution.

3.11.8 Simulation Techniques

Simulation can represent realistic systems for study of various scenarios that might occur over a time period at a modest cost. The structure of the shop, activities, jobs and constraints can be animated on a computer. Given appropriate input data and simple dispatching rules at decision points, computer could extrapolate a given schedule into the future. It provides a natural approach for interfacing with human expertise. However, the disadvantage is that the results obtained are not even approximately optimal and also it is difficult to determine how good these schedules are and how to improve them for better solutions. Simulation is the base for more advanced methods like Artificial Intelligence and Decision Support Systems with added accurate decision- making procedures.

3.11.9 Neighborhood Search Techniques

It is a general-purpose heuristic technique that may be used for quite complicated problems where solution itself is very complex. It consists of a starting solution called original seed and all solutions close to the original solution (the neighborhood of the seed). A selection criterion is used to find a new seed and this is terminated by a termination criterion. A much- improved solution is obtained at the end of the search.

3.11.10 Meta-Heuristic Search Methods

A meta-heuristic is a heuristic method for solving a very general class of computational problems by combining user-given black-box procedures- usually heuristics themselves-in the hope of obtaining a more efficient procedure. Meta-heuristics are generally applied to problems for which there is no satisfactory problem-specific algorithm or heuristic; or when it is not practical to implement such a method.

3.11.10.1 Tabu Search

The basic Concept of Tabu Search is described in *Glover* [23]. It is a deterministic heuristic approach for solving combinatorial optimization problems. It is an adaptive procedure that can be superimposed on many other methods to prevent them from being trapped at locally optimal solutions. It is a neighborhood search with a list of recent search positions. The essential feature of tabu search is the systematic use of memory. It keeps track of both the local information and also the exploration process. The method starts with an initial current solution, which could be feasible, non-feasible or even a partial solution. Using some local changes (called moves) from the current solution, a list of candidate solutions are generated (called candidate list). To avoid cycling in the algorithm a tabu list is maintained to keep track of a set of solutions that are forbidden. The role of the memory will be to restrict the choice to some subset of neighborhood by forbidding moves to some neighbor solutions.

3.11.10.2 Genetic Algorithm Approach

John Holland formally introduced genetic algorithms in the United States in the 1970s; Genetic algorithm derives its behavior from a metaphor of the processes of evolution in nature. This is done by the creation of a population of individuals represented by chromosomes, in essence a set of character strings that are analogous to the base-4 chromosomes that we see in DNA. The individuals in the population then go through a process of evolution. Genetic algorithms are used for solving multidimensional optimization problems in which the character string of the chromosome can be used to encode the values for the different parameters being optimized. In particular, genetic algorithms work very well on mixed (continuous and discrete), combinatorial problems. They are less susceptible to getting 'stuck' at local optima than gradient search methods. But they tend to be computationally expensive.

3.11.10.3 Simulated Annealing

As its name implies, the Simulated Annealing (SA) exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system. The current state of the thermodynamic system is analogous to the current solution to the combinatorial problem, the energy equation for the thermodynamic system is analogous to at the objective function, and ground state is analogous to the global minimum. The major difficulty (art) in implementation of the algorithm is that there is no obvious analogy for the temperature, T with respect to a free parameter in the combinatorial problem. Furthermore, avoidance of entrainment in local minima is dependent on the "annealing schedule", the choice of initial temperature, number of iterations performed at each temperature, and the temperature decrement at each step of cooling. Simulated annealing has been used in various combinatorial optimization problems and has been particularly successful in circuit design problems. The major advantage over other methods is the ability to avoid becoming trapped at local minima.

3.11.11 Moore -Hodgson's Algorithm

Moore-Hodgson's algorithm is an efficient and popular algorithm to minimize the number of tardy jobs in the single machine environment with ready times equal to zero. It is proved to be working well for the classical single machine problem scheduling problem with same objective function to minimize the tardy jobs. It is described further in detail in the Chapter 5.

CHAPTER 4

SINGLE MACHINE SCHEDULING

After getting the basic knowledge about scheduling theory in general and some initial background for handling scheduling problems, this chapter now describes introduction to single machine scheduling problems (referred to as SMS). It is organized as follows: Introducing the field of SMS research, then the importance and overview of SMS research is pointed out.

4.1 Definition, importance and overview of SMS Research

Definition "*The simplest pure sequencing problem is one in which there is a single resource, or machine*." There is not much more to say about the nature of single machine scheduling problems (also called one machine scheduling problems). n jobs , j = 1, ..., n (alternatively $j_j = j_1$, ..., j_n), are to be processed on one single machine (m = 1) under certain constraints.

At first - considering machine types and arrangements. The class of problems with parallel machines or dedicated machines is no longer relevant.

It is worth noting that when speaking about SMS problems of different perspectives are distinguished between two categories,

One is the category of lot size scheduling problems. In this case several different item types are to be processed on a single machine in lots or batches using repetitive production schedules (production cycles). The machine is typically restricted such that only one item type can be processed at a time and set-up costs (or set-up time) occur each time the processed type (more explicitly the batches or lots) is changed. These problems aim at determining the optimal batch sizes to minimize costs under certain constraints (for instance to meet a given demand of the items). *Bomberger* [9] mentions a metal stamping facility as an example. Stamps of different sizes and forms are produced on one press. Set-up time and costs occur each time a forming die must be changed.

The other is the category is that lot sizes are fixed and the aim is to determine a schedule specifying when each job is to be executed in order to achieve a certain objective.

Importance

Although single machine scheduling seems somehow obsolete at first glance - especially since real life problems are considered to be more extensive and difficult - it is still an important part in scheduling research. A short outlining of some reasons is given by *Baker* [1] and by *Gupta and Kyparisis* [24]:

SMS problems are often easier to understand and to handle mathematically than more comprising problems. This provides a valuable basis for a learning phase and for addressing general questions like getting familiar with the performance measures and for testing solution techniques. *Baker* [1] calls them *"a building block in the development of a comprehensive understanding of scheduling concepts."*

But not only for learning purpose, also for a deeper analysis of complex systems, an understanding of its incremental components, which often are nothing else than SMS problems, is essential.

In some cases, more complex scheduling issues can even fully be reduced to SMS related topics. In a multi-processor environment for example, focusing on a bottleneck or the most expensive machine might lead to a SMS problem which determines the schedule for the whole environment.

In a similar way, it might be appropriated - especially when handling a small production unit - to treat a complete production line as an aggregated single resource.

Focusing on more complex problem types with more than one machine in shop environments, single machine scheduling can be used as relaxation to obtain bounds.

In conclusion, the importance of SMS research is mainly based on two reasons: its simplicity on the one hand and the fundamental character for more complicated environments on the other.
Overview of SMS research single machine scheduling problems form the largest group within the area of scheduling research, comprise a variety of different settings and include a wide number of different constraints and objectives. They are commonly classified alongside the mentioned triple (three-field notation). Since most of the triple's instances can be combined, an almost endless number of problem types are possible.

4.2 Overview of Single Machine Scheduling Problems and their Complexity

Overview of single machining problems			
Specified problem type (complete classification)	Complexity	Method	Reference
$1\|\|C_{max}$	O(nlogn)	Each schedule that causes no idle times on the machine is optimal	
$1\|\|\sum C_j$	O(nlogn)	SPT-rule	*Smith*[49]
$[1\|\|\sum w_jC_j$	O(nlogn)	W SPT-rule	*Smith*[49]
$[1\|\|F_{max}$		Each schedule that causes no idle times on the machine is optimal	
$[1\|\|\sum F_j]$	O(nlogn)	SPT-rule	Equivalent to $[1\|\|\sum C_j]$
$[1\|\|w_jF_j$	O(nlogn)	W SPT-rule	Equivalent to $[1\|\|\sum w_jC_j]$
$[1\|\|w_{max}]$	O(n)	The job with longest processing time is to be scheduled last	
$[1\|\|\sum w_j$	O(nlogn)	SPT-rule	Equivalent to $[1\|\|\sum C_j]$
$[1\|\|\sum w_jW_i$	O(nlogn)	W SPT-rule	Equivalent to $[1\|\|\sum w_jC_j]$
$[1\|\|\sum L_{max}$	O(nlogn)	EDD-rule	*Jackson*[27]
$[1\|\|\sum w_jL_j)$	O(nlogn)	W SPT-rule	Equivalent to $[1\|\|\sum w_jC_j]$
$[1\|\|\sum L_j]$	O(nlogn)	EDD-rule	Equivalent to $[1\|\|\sum C_j]$
$[1\|\|T_{max}]$	O(nlogn)	EDD-rule	*Jackson*[27]
$[1\|\|\sum T_j]$	NP-Hard		*Du and Leung*[22]
$[1\|\|\sum w_jT_j$	NP-Hard		*Lawler*[38]
$[1\|\|\sum U_j$	O(nlogn)	Hodgson Algorithm	*Moore*[45]
$[1\|\|\sum r_jU_j$	NP-Hard		
$[1\|\|w_jU_j$	NP-Hard		*Karp*[30]

Table 4.1: Complexity of elementary SMS problems

Table 4.1 shows overview - including the research results and directions for the main classes - is given by *Gupta and Kyparisis* [24]. Starting with a tree-like classification presentation, they review SMS research within the span of time between its upcoming in the mid 1950s to the late

1980s. Their research effort is limited to static SMS problems - as a remainder those with each job's release time is zero. They further do not consider any stochastic behavior.

Minimizing maximum makespan [1||C_{max}], maximum flow time [1||F_{max}]

A Minimization of the makespan equals minimizing the idle time of the machine. Thus, each schedule that causes zero idle times is optimal. Since the flow time is defined as $F_j = C_j - r_j$, both problems are the same if $r_j = 0 \; \forall \; j$.

Minimizing total completion time [1||$\sum C_j$], total flow time [1||$\sum F_j$], total waiting time [1||$\sum W_j$] and total lateness [1||$\sum L_j$]

Considering the general case of m machines, a decomposition of the flow time leads to following relationships:

$$F_j = \sum_{1}^{m} (W_{ji} + p_{ji}) = W_j + \sum_{1}^{m} p_{ji} \text{ and } F_j = C_j - r_j = d_j + L_j - r_j.$$

Hence, $\sum C_j$, $\sum F_j$, $\sum W_j$ and $\sum L_j$ are equivalent objective functions since each can be modified to one of the others via linear-transformation (multiplication and / or addition with constant parameters). *Smith* [49] showed the optimality of the SPT-rule ("always serving the job with the shortest processing time first") to minimize the total completion time, thereby implicitly including an optimal method for the other problems. The method is of complexity O (n log n).

Minimizing total weighted completion time [1||$\sum w_j C_j$], total weighted flow time [1||$\sum w_j F_j$], total weighted waiting time [1||$\sum w_j W_j$] and total Weighted lateness [1||$\sum w_j L_j$]

With the use of the relationships shown above, the equivalence of the objectives $\sum w_j C_j$, $\sum w_j F_j$, $\sum w_j W_j$ and $\sum w_j L_j$ becomes obvious

$$\sum w_j F_j = \sum w_j W_j + \sum_{j=1}^{n} w_j \sum_{i=1}^{m} p_{ji} \text{ and}$$

$$\sum w_j F_j = \sum w_j C_j - \sum_{j=1}^{n} w_j r_j = \sum w_j L_j + \sum_{j=1}^{n} w_j (d_j - r_j).$$

It was again *Smith* [49] who proved optimality of the WSPT-rule ("prioritizing according to the lowest value of the ratio of processing time to weight") for the total weighted completion time and hence for the other listed problems. This method is polynomially bounded by O (n log n).

Minimizing maximum waiting time [1||W_{max}] can be solved in O (n) by simply scheduling the job with the largest processing time last.

Minimizing maximum tardiness and maximum lateness, i.e. [1||T_{max}] and [1||L_{max}], respectively

Jackson [27] applied the EDD-rule ("serving the job with the earliest due date first") to solve both problems with computational effort of O (n log n).

The complexity of the problem of **minimizing the total tardiness [1||$\sum T_j$] r**emained open for a long time until *Du and Leung* [22] proved NP-hardness.

Considering the **minimization of weighted tardiness [1||$\sum w_j T_j$],** which is the more general version of the problem mentioned before, it was *Lawler* [38] and *Lenstra et al*. [40] who showed NP-hardness.

The problem of **minimizing the number of tardy jobs with release time constraints [1|r_j|$\sum U_j$]** is NP-hard.

The problem of **minimizing the weighted number of tardy jobs [1||$\sum w_j U_j$]** . Karp [30] showed NP-hardness remaining Even if some jobs have a common due date.

Above special case with unit weights, i.e. minimizing the number of tardy jobs [1||$\sum U_j$], can be solved in polynomial time O (n log n) by applying the Algorithm of *Moore* [45] (also hodgson or Moore-Hodgson algorithm).

Treated problem types

This thesis' work is dedicated to SMS research review dealing with problems that include due dates. That means: a due date d_j is assigned to each job j determining the point in time at which the job should be completed at the latest.

When dealing with due dates, two cases become obvious. A job can be finished before its due date - in this case one speaks of earliness:

$$E_j = \max \{0, d_j - C_j\}.$$

On the other hand, finishing a job can be delayed causing a completion time that exceeds the due date - this is what is referred to as tardiness:

$$T_j = \max \{0, C_j - d_j\}.$$

Focusing not on the absolute time a job is delayed but only on the fact, a binary variable U_j is introduced indicating whether a job exceeds its due date or not.

$$U_j = \begin{cases} 1, & if\ T_j > 0, \\ 0, & T_j = 0 \end{cases}$$

CHAPTER 5

MINIMIZING THE NUMBER OF TARDY JOBS

5.1 Problem Presentation

This chapter addresses the problem $[1|...|U_j]$, i.e. minimizing the number of tardy jobs on single machine subject to certain job characteristics. n jobs, j = 1, ..., n (alternatively $J_j = J_1, ..., J_n$), are to be scheduled on one single machine. The job's processing time is p_j, its release date r_j and its duedate d_j, with $r_j + p_j \leq d_j$. The job is tardy ($U_j = 1$) if its completion time exceeds its due date ($C_j > d_j$), otherwise it is on time.

A property often referred to shall be prefixed is that if a job j is tardy it might as well be arbitrarily tardy, meaning that it can be scheduled arbitrarily after all on-time jobs. Thus, an optimal schedule exists such that all jobs on time precede all tardy jobs. And the objective of "minimizing the number of tardy jobs" is equivalent to "maximizing the number of on-time jobs".

This chapter is organized as following:

Starting with Section 5.2, a detailed literature review is given.

Further, Section 5.3 addresses some "traditional" solution approaches. The well known Hodgson algorithm as the basic optimal method for the elementary problem $[1||\sum U_j]$ is introduced, as well as an approach by *Kise et al* [31] for $[1|r_j|\sum U_j]$ with similarly ordered release dates and due dates.

5.2 Historical Development and Research overview

Moore [45] proposed an O (n log n) polynomial algorithm for the most general problem $[1||\sum U_j]$ by consequently applying the EDD-rule. Later on, T. J. Hodgson suggested a simpler version of this algorithm and gave this algorithm his name. Notwithstanding, the algorithm gained popularity widely known as Moore or Moore-Hodgson algorithm. *Maxwell* [42] proposed an integer programming formulation solvable with an algorithm revealing Moore's algorithm. An extension to the case where a specified subset of jobs must necessarily be on time is provided by *Sidney* [48]. Under more restrictive assumptions concerning the jobs characteristics, the problem $[1|r_j|\sum U_j]$ is proven to be NP-Hard.

But when release dates and due dates are similarly ordered ($r_i \leq r_j =$) $d_i \leq d_j \rightarrow \forall i, j$) the problem is solvable in O (n^2) polynomial time by applying the dynamic programming algorithms proposed in *Kise* et al.[31] and *Lawler*[36]. Some years later, the complexity for the problem with similarly ordered release and due dates was reduced to O (n log n) by *Lawler* [38] using a modification of the Moore-Hodgson algorithm.

Based on the work by *Lasserre and Queyranne* [33], several mixed integer linear programming formulations (MILP) emerged. By relaxing binary constraints in MILP formulations, *Dauz`ere-P´er`es* [15] and *Dauz`ere-P´er`es* [16] obtain lower bounds. Due to the size of the MILP program, no more than 10 jobs can be considered making these approaches unsuited for an implementation in branch and bound procedures.

More effective bounds can be obtained by relaxing the non-preemption constraint - permitting interruptions. For the latter problem, referred to as [1|pmtn, $r_j|\sum U_j$], several dynamic programming algorithms are known. *Lawler* [37] proposed a procedure bounded by O (n^5) if all release dates are distinct, whereas the algorithm proposed by Baptiste [3] is polynomially bounded by O (n^4) and thus provides the lowest complexity.

Another interesting approach to achieve a lower bound is to relax due dates such that the time-windows ($d_j - r_j$) become nested $[r_1, d_1] \leq [r_2, d_2] \leq \ldots [r_n, d_n]$. For this case ($[1|r_j$, nested$|\sum U_j]$), Briand et al.[10] introduced an O (n^3) algorithm.

Carlier [11] proposed an O (n^3 log n) polynomially bounded algorithm for the problem with equaling processing times: $[1|r_j, p_j = p|\sum U_j]$. More than two decades later, *Chrobak et al*.[13] proved that this algorithm may produce suboptimal results, and they presented an O (n^5) dynamic programming algorithm by their own. Four different recent branch and bound procedures for $1|r_j|\sum U_j]$ attract attention. *Dauz`ere-P´er`es and Sevaux* [18] suggest a B&B method based upon the notion of a master sequence, i.e. a sequence from which it is known that it contains at least one optimal solution.

Baptiste et al [6] consequently use elimination rules and dominance properties in order to quicken the search procedure. Another interesting approach by *Baptiste et al*.[6] interprets the problem as a constraint satisfaction problem and embeds well known constraint propagation techniques into a B&B procedure.

Recently, *M'Hallah and Bulfin* [43] published a B&B approach for the weighted case, which is also capable of solving the non-weighted case. They formulate $[1|r_j \sum U_j]$ as a linear program and use surrogate relaxation to obtain a multiple-choice knapsack problem for which B&B algorithms are well known.

Another work worth mentioning, since it is one of the few that considers certain precedence constraints among jobs, was presented by *Ibarra and Kim* [26], who developed a heuristic for this special problem.

As already mentioned, this thesis is dedicated to deterministic scheduling, meaning that the job's characteristics, such as processing times, release dates and due dates are known for sure, which is an outrider to stochastic scheduling, with the objective of minimizing the number of tardy jobs *Balut* [2]. A short survey of scientific efforts in that area is given by *Jang and Klein* [28].

Table 5.1 gives an overview of the field of research.

5.3Similar Problems in the Past

Many of the researcher's have studied the problem in the past and obtained various solution approaches. The table provides the history information of the problem. The table shows the specified problem, the developed year and the solving method.

Number of Tardy Jobs						
Specified Problem	Year	Author	Solving Method	Character	Complexity	Comment
$[1\|\|\sum U_j$	1968	*Moore*	Moore's Algorithm	op	O(nlogn)	
$[1\|\|\sum U_j]$	1970	*Maxwell*	Moore's Algorithm	op	O(nlogn)	
$[1\|\|\sum U_j]$	1978	*Kise et al.*	DP	en	$O(n^2)$	$r_i<r_j \rightarrow d_i \leq d_j$
$[1\|\|\sum U_j]$	1995	*Dauzere-Peres*	Heuristic	he	$O(n^2)$	
$[1\|r_j,p_j=p\|\sum U_j]$	1981	*Carlier*	DP	en	$O(n^3 \log n)$	
$[1\|r_j\|\sum U_j]$	1982	*Lawler*	DP	en	$O(n^2)$	$r_i<r_j \rightarrow d_i \leq d_j$
$[1\|r_j\|\sum U_j]$	1990	*Lawler*	DP	en	$O(n^2)$	$r_i<r_j \rightarrow d_i \leq d_j$
$[1\|r_j\|\sum U_j]$	1994	Lawler	Generalization of Moore's Algorithm	op	O(nlogn)	$r_i<r_j \rightarrow d_i \leq d_j$
$[1\|r_j\|\sum U_j]$	1995	*Dauzere-Peres*	Linear programming relaxation	he		Lower bound calculation
$[1\|r_j\|\sum U_j]$	1997	*Dauzere-Peres*	Linear programming relaxation			
$[1\|r_j\|\sum U_j]$	1998	*Baptise et-al*	B&B	en		Constraint Propagation technique are used
$[1\|r_j\|\sum U_j]$	1999	*Dauzere-Peres & Sevaux*	B&B	en		
$[1\|r_j\|\sum U_j]$	2003	*Baptise et-al*	B&B	en		
$[1\|r_j,p_j=p\|\sum U_j]$	2004	*Chroback et al*	DP	en	$O(n^5)$	
$[1\|r_j\|\sum U_j]$	2007	*M'Hallah and Bulfin*	B&B	en		
$[1\|pmtn,r_j\|\sum U_j]$	1990	*Lawler*	DP	en	$O(n^3k^2)$ with	

					k as number of distinct release dates	
$[1\|pmtn,r_j\|\sum U_j]$	1994	*Lawler*	Generalization of Moore's Algorithm	op	O(nlogn)	$r_i<r_j \rightarrow d_i \leq d_j$
$[1\|pmtn,r_j\|\sum U_j]$	1999	*Baptise et-al*	DP	en	$O(n^4)$	
$[1\|r_j\|,nested\|\sum U_j]$	2006	*Briand et al.*	Algorithm	op	$O(n^3)$	
$[1\|prec,r_j\|\sum U_j]$	1978	*Ibarra &Kim*	Heuristic	he		
DP=Dynamic Programming B&B=Branch and Bound ,op=optimal ,en=enumerative, he=heuristic						

Table 5.1:-History

5.4 Traditional Solution Approaches

Two traditional, well-known algorithms for minimizing the number of tardy jobs on a single machine are outlined. Both approaches are quite simple since they are subject to narrow restrictions. Thus, in most cases, they do not picture the reality very well. But notwithstanding, they offer a good introduction and can be used as lower bounds for more complex problems. For this, Section 5.4.1 describes the well known Hodgson algorithm followed by 5.4.2 pointing out an algorithm proposed by *Kise* et al.[31]. Especially the first approach is of much importance since several other special cases can be solved by slight modifications.

5.4.1 The Basic Problem - the Hodgson Algorithm

Moore [45] introduced an algorithm capable of scheduling n jobs on a single machine with known processing times p_1, ..., p_n and due dates d_1, ..., d_n. Set-up times are included in the sequence independent processing times. Furthermore, preemption is not permitted and all jobs are available for processing throughout the whole process. Concluding, the problem is characterized as $[1||\sum U_j]$.

It is assumed that all jobs can be completed on time if they are processed immediately at time t = 0; i.e. $p_j \leq d_j \ \forall j$. Jobs not satisfying this assumption can be eliminated for reducing the problem size.

The algorithm is based on the division of the set of jobs into two subsets (or sequences). The jobs in the first sequence O - those that are to be scheduled on time - are ordered in the direction of ascending due date (the EDD-rule) whereas the second sequence, L, contains the jobs deemed to be tardy. Those can be scheduled in arbitrary order. If in a final sequence O is scheduled before L, the maximum number of jobs is on time, or equivalently the minimum number of jobs is tardy. The following abbreviations are used:

• J is the sequence containing all jobs.

• O is a sequence containing on-time jobs.

• L is the set or sequence of arbitrarily ordered tardy jobs.

• t is the currently viewed point in time.

In the following, the algorithm is presented in detail. The first step is the sorting of jobs in J according to the EDD-rule. Thereafter, one job after the other is removed from J and sequenced in O as long as the total processing time of O is less than the due date of the last job in O. Does the latter condition not hold, the job with the longest processing time in O is deferred to L and the completion time is adjusted. This procedure is repeated until the sequence J is empty and all Jobs are either part of sequence O or L. The final sequence is optimal if O is scheduled before L.

Algorithm 5.4.1.1

The Moore-Hodgson algorithm for the basic problem

Step 1: Order all jobs in the set J using EDD rule.

Step 2: If no jobs in J are late; J must be optimal and go to Step 4. Otherwise, find the first late job in J. Let this first late job be the k^{th} job in set J.

Step 3: Out of the first k jobs, find the longest job. Remove this job from J and put it in L. Return to Step 2.

Step 4: Form an optimal sequence by taking the current sequence of jobs in O and appending rejected jobs in set L in any order.

The Hodgson algorithm solves $[1||\sum U_j]$ optimally and is, due to the sorting according to the EDD-rule, polynomially bounded by O (n log n).

Hodgson's algorithm can be applied to $[1||\sum w_j U_j]$, the corresponding weighted problem, if processing times and weights can be indexed such that an opposite ordering is obtained ($p_i < p_j \rightarrow w_i \geq w_j \; \forall i, j$). Further modifications are possible for a bunch of other variations of the "weighted number of tardy job" problems.

5.4.2 Similarly Ordered release and due dates – the Kise et al.[31] Algorithm

Kise et al [31] propose a dynamic programming algorithm (in the sequel referred to as Kise et al. algorithm) for $[1|r_j|\sum U_j]$ under the additional assumption that release and due dates can be indexed such that a similar order is obtained:

$$r_1 \leq ... \leq r_n \quad \text{and} \quad d_1 \leq ... \leq d_n. \tag{5.1}$$

Later on, *Lawler* [36] and *Lawler* [38] also proposed dynamic programming algorithms solving the same problem. While the approaches by Lawler [36] and Kise et al.[31] have a complexity of O (n^2), *Lawler*[38] reduced the computational effort to O (n log n). As proved in *Dauz`ere-P´er`es and Sevaux* [17], condition (5.1) can be relaxed to

$$r_i \leq r_j \rightarrow \begin{cases} d_i \leq d_j, & or \\ r_j + p_j + p_i > d_i & \forall i, j \end{cases} \tag{5.2}$$

The problem discussed at this point contains the problem $[1||\sum U_j]$ as a special case. Since when all release dates are equal, i.e. $r_1 = ... = r_n$, condition (5.1) is self-evident.

Like Hodgson's algorithm, the Kise et al. algorithm decomposes the set J of n jobs into two subsets called O and J-O. In the final schedule all jobs in set O are sequenced according to their indexes (equation (5.1)), i.e. in EDD-order, and the jobs in subset J-O are sequenced arbitrarily. This leads to the concept of optimality stated by Lawler [34]. According to the latter concept, "a set E_j is j-optimal if it is the set of early jobs in an optimal schedule for the set of jobs $\{1,\ldots, j\} \subset J$".

Analogically, a set E_n is n-optimal if it is a set of early jobs in an optimal schedule (E_n, $J-E_n$) for J. The early, n optimal subset E_n is scheduled - with jobs in EDD-order - in front of the set containing the tardy jobs ($J - E_n$). Consider a simple example with 5 jobs ($J = \{1, 2, 3, 4, 5\}$) complying with the assumption in equation (5.1) and an optimal schedule where the jobs are sequenced in ascending order of their indexes: If, in the optimal schedule, the first three jobs are early and the others tardy, then $E_5 = \{1, 2, 3\}$.

For the subset $E_j = \{1, ...,m\}$ with ($1 < ... < m$ and $m \leq j$), the completion time of certain job i, $C_i(E_j)$, in a j -optimal set can be computed as follows:

$$C_i(E_j) = \max \{C_{i-1}(E_j), r_i\} + p_i. \tag{5.3}$$

Of vital importance for the algorithm is the distinction of two cases. If job j is scheduled immediately after E_{j-1}, the j -optimal set up to the current state in the algorithm, and is finished on time, then job j becomes part of E_j , i.e. $E_j = E_{j-1} \cup \{j\}$. Otherwise, job j only becomes part of E_j if a change for some job l ($l \in E_{j-1}$) shortens the processing time. This distinction is mapped as follows:

$$E_j = \begin{cases} E_{j-1} \cup \{j\}, & if\ C(E_{j-1} \cup \{j\}) \leq d_j, \\ E_{j-1} \cup \{j\} - \{l\}, & otherwise \end{cases} \tag{5.4}$$

Where $E_0 = ¢$ is assumed and l is a job satisfying

$$C(E_{j-1} \cup \{j\} - \{l\}) \leq C(E_{j-1} \cup \{j\} - \{i\}), \forall i \in E_{j-1} \cup \{j\}. \quad (5.5)$$

For a better understanding the algorithm is divided into one main algorithm and one Sub-algorithm. The main algorithm is quite comprehensible, while the sub-algorithm is complex and requires additional elucidations. The main algorithm computes E_j using Algorithm 5.4.2.2

Kise et al.[41]: Main algorithm 5.4.2.1

1: start $E_0 = \phi$, $j \leftarrow 0$
2: while $j < n$ do
3: $j \leftarrow j + 1$
4: if $C(E_{j-1} \cup \{j\}) \leq d_j$ then
5: $E_j = E_{j-1} \cup \{j\}$
6: else
7: execute Sub algorithm 5.4.2.2 {to compute l}
8: $E_j = E_{j-1} \cup \{j\} - \{l\}$
9: end if
10: end while
11: $(E_n, J - E_n)$ is an optimal schedule

Equations (5.4) and (5.5), if $C(E_{j-1} \cup \{j\}) \leq d_j$ holds, the calculation of E_j is easy and done in constant time. If this is not the case, the sub algorithm is called for computing job l.

The sub algorithm generates for the current $E_{j-1} = \{1,..., m-1\}$, with job j equaling

m, two sets of jobs, S_i and S_i':
$S_i = \{1,...,h-1, h+1, ..., i\}$, (5.6)
$S_i' = \{1,....,i-1\}$, (5.7)
for i = 1, ...,m, together with their associated processing times $C(S_i)$ and $C(S_i')$ and with l = h satisfying

$$C(S_i) \leq C(S_i'), i = 1, ...,m. \quad (5.8)$$

Upon termination (i= m), the excluded l = h becomes equal to l in equation (5.5).

Algorithm 5.4.2.2 Kise et al.[31]: Sub-algorithm

1: start m = j
2: if m = 1 then

3: $S_i \leftarrow \emptyset$; $l \leftarrow m$
4: halt
5: else
6: $i \leftarrow 2$
7: $S_i \leftarrow \{2\}$
8: $C(S_i) \leftarrow r_2 + p_2$
9: $S_i' \leftarrow \{1\}$
10: $C(S_i') \leftarrow r_1 + p_1$
11: $l \leftarrow 1$
12: while $i \leq m$ do
13: if $C(S_i) > C(S_i')$ then
14: $S_i \leftarrow S_i'$
15: $C(S_i) \leftarrow C(S_i')$
16: $l \leftarrow i$
17: end if
18: if $i < m$ then
19: $S_{i+1} \leftarrow S_i \cup \{i + 1\}$
20: $C(S_{i+1}) \leftarrow \max\{C(S_i), r_i\} + p_i$
21: $S_{i+1}' \leftarrow S_i' \cup \{i\}$
22: $C(S_{i+1}') \leftarrow \max\{C(S_i'), r_i\} + p_i$
23: end if
24: $i \leftarrow i + 1$
25: end while
26: end if

CHAPTER 6

PROBLEM STATEMENT AND METHODOLOGY

In the last chapter we discussed the research done on minimizing number of tardy jobs scheduling problems, the solution methodologies applied and the complexity of those algorithms. In this chapter, the present research problem and our methodology to solve it will be presented.

6.1 Statement of the Problem

The problem in the present research is, scheduling of n jobs that are available at time r_j on a single machine to minimize the number of tardy jobs. Each job is associated with a release time, processing time and due date. The machine can perform one operation at a time and no preemption is allowed. A job is said to be tardy when its completion time is greater than the due date associated with it.

6.2 Assumptions

1. The machine is always available

2. No preemption is allowed.

6.3 Objective Function

The objective function that is been considered for the present problem is to minimize the number of tardy jobs.

Minimize$\sum U_j$, for j = 1, 2,.., n

Where $\mathbf{U_j} = \begin{cases} 1, & if\ job\ \ j\ \ is\ tardy \\ 0, & otherwise \end{cases}$

6.4 Complexity of the Problem

The $1 \mid r_j \mid \sum U_j$, is proved to be a NP-hard problem and an optimal solution to this problem can be found only in pseudo-polynomial time. We devised a heuristic algorithm and branch and bound algorithm for the above problem and obtained result from both are compared.

.

6.5 Methodology

6.5.1 A Heuristic

This section introduces a heuristic again proposed by Dauz`ere-P´er`es [15]. After an introductory description, an estimation of the heuristic's performance is given by comparing its solutions to B&B algorithm.

J denotes the set of jobs to be scheduled, with $J_i = J_1, ..., J_n$ pointing at a typical job. S_i marks the actual starting time of a job J_i, with $s_i \geq r_i$ ensuring that J_i is not processed before its release date. The completion time of J_i is noted as C_i and can be expressed as $C_i = s_i + p_i$.

The heuristic divides J, the set of jobs to be scheduled, into three subsets O, L and F in the way that $J = O \cup L \cup F$, with:

O being a set of jobs scheduled early (on time),
L being a set of tardy jobs,
F being a set of outstanding (free) jobs (the jobs belonging to F are not scheduled at a special point during the heuristic), and

P (O) denoting the sequence associated to O,
C (O) denoting the completion time of the last job in P (O).

O is just a set of jobs not revealing anything about the sequence in which the jobs associated with O are scheduled. This sequence is determined by P (O), whereas C (O) denotes the time needed to process all jobs in O, which of course depends on P (O). Let J_k be the last job scheduled in P(O), then C(O) can be computed to $C(O) = s_k + p_k$. The heuristic's principle is as follows: Amongst the jobs in F the job J_j with the smallest duedate is chosen and relocated to O. If this job becomes tardy after its relocation ($C_j = s_j + p_j > dj$), a job $J_l \in O$ satisfying the condition
$C(O - \{J_l\}) = \min_{Ji \in O} C(O - \{J_i\})$

is appointed and relocated to L in the first "if - then" construct. The latter condition simply ensures that the particular job J_l is chosen and relocated such that its absence from O minimizes the completion time of O.

Let the index k point at the last job in the current sequence of O, with J_k marking this particular last job. In the second "if - then" construct the job $J_l \in L$ with the smallest processing time amongst those ready for processing is selected and exchanged for J_k if the chosen J_l becomes early ($r_l \leq s_k$ and $s_k + p_l \leq d_l$) and the last job's completion time decreases ($p_l < p_k$). The aim of reducing the completion time in both "if - then" constructs is to liberate the machine as soon as possible for the next selected job.

Algorithm 6.5.1.1 A heuristic to [1|r_j |$\sum U_j$]

Set $t = \min_{J_i \in J} r_i$, O = L =¢; and F = J

while F≠¢ do

Among the jobs in F such that $r_i \leq t$, choose J_j with the smallest due date

$O = O \cup \{J_j\}$, $F = F - \{J_j\}$, $s_j = t$, and $C(O) = C_j = s_j + p_j$
if $C(O) > d_j$ then
Search $J_l \in O$ such that $C(O - \{J_l\}) = \min_{J_i \in O} C(O - \{J_i\})$
$O = O - \{J_l\}$, $L = L \cup \{J_l\}$, update s_k
$C(O) = s_k + p_k$ {J_k last job in P (O)}
end if
if there are jobs in L such that $r_i \leq s_k$, and $s_k + p_i \leq d_i$ then
choose the one with the smallest processing time to be J_l
if J_l exists and $p_l < p_k$ then
$O = O - \{J_k\}$, $O = O \cup \{J_l\}$, $L = L - \{J_l\}$, $L = L \cup \{J_k\}$
update s_l, $C(O) = s_l + p_l$
end if
end if
$t = \max \{C(O), \min_{J_i \in F} r_i\}$
end while

Computational results numerical experiments show the optimal solution on Sun 4.1 workstation (the paper was published in 1995), obtained by applying algorithm 6.5.1.1 and branch and bound algorithm for an experiment with 10 jobs

Problem number	1 2 3 4 5 6 7 8 9 10
Heuristic	6 7 7 5 5 6 6 5 8 6
Branch and Bound	6 7 7 5 5 6 6 5 8 6

Table 6.1: Comparing the heuristic (Algorithm 6.5.1.1) and branch and bound procedure

Its performance seems to be quite good, but here is the conducted experiment considers only 10 jobs.

6.5.2 A Branch and Bound Procedure Based on a Master Sequence

This section describes a B&B approach proposed by Dauz`ere-P´er`es and Sevaux [18] for the problem $[1|r_j|\sum U_j]$. Based on the concept of a master sequence, a sequence which contains at least one optimal solution, it was primary developed for the weighted case ($[1|r_j|\sum w_j U_j]$) but can also be used if weights are neglected. Starting with the description of the concept of a master sequence, a small example for a better understanding to generate master sequence is given.

6.5.2.1 Master Sequence

As already mentioned, the master sequence is a sequence from which it is proven that at least one optimal sequence can be derived. The following theorem builds the basis for the generation of such sequence:

Consider a set of n jobs, "there is always an optimal sequence of jobs on time that solves the problem $[1|r_j|\sum w_j U_j]$, in which every job J_j is sequenced just after J_i [(noted as (J_i, J_j))] such that" one of the following conditions holds:

$d_i < d_j$, or, (6.1)
$d_i \geq d_j$ and $r_k \leq r_j$, $\forall J_k$ sequenced before J_j . (6.2)

It is easily said that not satisfying both conditions (6.1) and (6.2) causes following condition to hold:

$d_i \geq d_j$ and $\exists$ J_k sequenced before J_j such that $r_k > r_j$. (6.3)

The proof of the theorem is by showing that any optimal sequence which minimizes the number of tardy jobs can be manipulated in the way that either condition (6.1) or (6.2) is satisfied.

Suppose having an optimal sequence in which for some (or all) jobs condition (6.3) is satisfied. Consequently, the latter sequence is not compatible with a master sequence.

Now identify the first pair of jobs (J_i, J_j) satisfying the latter condition and their associated starting times s_i and s_j, respectively. If one interchanges both jobs, J_j is allowed to start at s_i since $\exists J_k$ sequenced before J_j such that $r_j < r_k \leq s_i$. Further, after the interchange, J_i starts at $s_i + p_j$ and will still be finished on time since $s_i + p_i + p_j \leq d_j \leq d_i$. This interchange is continued until either condition (6.1) or (6.2) is satisfied for J_j and its new predecessor. If one cannot identify a predecessor satisfying one of these conditions, J_j has to be scheduled first. This procedure is repeated until all jobs in the optimal sequence satisfy the necessary conditions. To clarify the

proof's procedure, consider a set of three jobs J_1, J_2 and J_3. The associated processing times, due and release dates are outlined in the subsequent Table 6.2.

Job	J_1	J_2	J_3
p_i	1,5	1,5	1
r_i	1	2	1
d_i	4	5	5

Table 6.2: Transforming an optimal sequence into the master sequence – example

Obviously, the sequence (J_1, J_2, J_3) is feasible and at the same time an optimal solution. When looking at the pair (J_1, J_2), condition (6.1) is satisfied. But when checking the pair (J_2,J_3), condition (6.1) is violated, as well as condition (6.2), since there is one job (in that case J_2) sequenced before J_3 with a release date larger than r_3. After changing J_2 and J_3, one reaches the sequence (J_1, J_3, J_2). Again all jobs are finished on time and either condition (6.1) or (6.2) is satisfied.

In the following, σ denotes the master sequence and S a subset of sequences satisfying either conditions (6.1) or (6.2). Every sequence of S is contained in the master sequence. Note that in the sequel only sequences belonging to S will be considered since it is known that this subset comprises at least one optimal solution. The master sequence (σ) can be computed by applying Algorithm 6.5.2.1.1, which is polynomially bounded by O (n^2). Let J denote the initial set of all jobs and let these jobs be ordered according to non-decreasing release dates. Additionally, a set $\bar{J}$ containing the already sequenced jobs ordered in non-decreasing order of their due dates is maintained during the algorithm.

Algorithm 6.5.2.1.1 Algorithm to calculate the master sequence

1: jobs in J are supposed to be ordered in non-decreasing order of their release dates
2: for all $J_i \in J$ do
3: $\sigma \leftarrow \sigma \cup J_i$
4: $\bar{\jmath} \leftarrow \bar{\jmath} \cup J_i$ {Jobs ordered in non-decreasing order of their due dates}
5: for all $J_j \in \bar{\jmath}$, $J_j \neq J_i$ such that $d_j \geq d_i$ do
6: $\sigma \leftarrow \sigma \cup J_i$
7: end for
8: end for

When looking at the algorithm, it becomes obvious that particular jobs can occupy multiple positions in the master sequence. Every job is first added to the master sequence by the operation in line (3) and to $\bar{\jmath}$ (line (4)). The associated position in σ is called a generating position. Afterwards, some $J_j \in \bar{\jmath}$, which are already part of the master sequence, are added again to the master sequence if these jobs have a due date larger than the last added job. Note that these jobs are added in non-decreasing order of their due dates. The corresponding positions generated by line (5) to (7) in Algorithm 6.5.2.1.1are called generated positions. The following example stated in Table 6.3 clarifies the latter explanations.

Job	J_1	J_2	J_3	J_4	J_5
r_i	12	15	23	37	47
p_i	10	92	66	57	3
d_i	34	142	105	147	76

Table 6.3: Generating the master sequence - example

In Table 6.3 the jobs are already ordered in non-decreasing order of their release dates. The algorithm starts by adding J_1, J_2, J_3 to both σ and $\bar{\jmath}$ in the first three runs. These are generating positions. After adding J_3, the second "for" loop recognizes that there is a job in $\bar{\jmath}$ with a due date larger than the job's due date in the last generating position ($d_2 > d_3$), and thus J_2 is added again to. The latter position is a generated position. Afterwards, J_4 and J_5 are added in a generating position before the second "for" loop intervenes again and adds J_3, J_2, J_4, since $d_5 < d_3 < d_2 < d_4$. Therefore, the resulting master sequence is:

$\sigma = (J_1, J_2, J_3, J_2, J_4, J_5, J_3, J_2, J_4)$.

Now, with known σ, it is easy to derive S. This is done by extracting - by selecting - for each job at most one position of σ. Selecting ensures that the resulting sequence is actually compatible with σ meaning that the relative sequence of any sequence pertaining to S mirrors the master

sequence's relative order of jobs. Glancing back at the example with σ = (J_1, J_2, J_3, J_2, J_4, J_5, J_3, J_2, J_4) as master sequence, a subset of S containing all 5 jobs is stated below:

{(J_1, J_2, J_3, J_4, J_5), (J_1, J_2, J_3, J_5, J_4), (J_1, J_2, J_4, J_5, J_3), (J_1, J_2, J_5, J_3, J_4),
(J_1, J_3, J_2, J_4, J_5), (J_1, J_3, J_2, J_5, J_4), (J_1, J_3, J_4, J_5, J_2),
(J_1, J_3, J_5, J_2, J_4), (J_1, J_4, J_5, J_3, J_2), (J_1, J_5, J_3, J_2, J_4)}.

It is obvious that such sequences satisfy either condition (6.1) or (6.2). Of course, other sequences that include less than 5 jobs are possible.

6.5.2.2 The Branching Scheme

The branching is executed on the positions in the master sequence and obeys the last-in- first-out rule. Starting on the first position in the master sequence it progresses gradually. Each node generates two further nodes in the following way:

The first node (left branch):

Sequence - set on time - the next job in the master sequence (σ) and update the completion time of the new partial sequence. Remove all remaining positions of the just sequenced job from σ.

Not yet sequenced jobs which are already tardy are removed from the master sequence together with their corresponding positions.

The second node (right branch):

The next position in the master sequence is not sequenced but removed.
If this is the first position of the associated job in the master sequence, it is a generating position. Eventually associated generated positions are skipped.

The associated job must be tardy if it seizes no further position in the master sequence.

Note that in every step some positions in the master sequence might be removed. Due to only removing positions and corresponding jobs from the master sequence, each partial sequence in each node belongs to S and satisfies either condition (6.1) or (6.2).

6.5.2.3 The Bounding Scheme

After the determination of the master sequence, the heuristic Algorithm is used to compute an upper bound. The latter heuristic is executed only once at the beginning of the B&B procedure. It is polynomially bounded by O (n^2). The upper bound is updated in the course of the B&B

procedure as soon an admissible lower bound, better than the current upper bound, is calculated. Additionally, at the root of the branching relaxations are proposed to calculate lower bounds. The bound is obtained by relaxing the duedate or release date such that, in cases, $r_i \leq \ldots \leq r_n$ and $d_i \leq \ldots \leq d_n$ holds.

This special problem is polynomially solvable with the well known algorithm proposed by Kise et al.[31] in O (n^2) steps. After applying the Kise et al. algorithm on relaxations, the lower bound is obtained. Both relaxations, followed by the mentioned algorithm, can also be applied to evaluate every node in the branching tree, i.e. to compute LB's. Therefore, relaxing the due or the release dates of those remaining jobs that are not scheduled yet in the master sequence is necessary. The calculation of the latter lower bounds at each node can be done in O (n^2). Baptiste et al. [6] propose a short procedure which adjusts the duedates in O (n log n) steps before the Kise et al. algorithm can be executed

Algorithm 6.5.2.3.1 Relaxation of due dates

Sort jobs in non-decreasing order of their release dates
$d_{max} \leftarrow 0$
for $J \leftarrow 1$ to J_n do
if $d_i < d_{max}$ then
$d_i \leftarrow d_{max}$
else
$d_{max} \leftarrow d_i$
end if
end for

Apply Kise et al.[31] algorithm

Note that by sorting the jobs in non-decreasing order of their due dates and adjusting the release dates in the "for" loop, rather than the due dates, the procedure can also be used for the relaxation of the release dates.

CHAPTER 7

COMPUTATIONAL EXPERIMENTS AND RESULTS

In the last chapter the heuristic solution approach and the branch and bound procedure based on master sequence to solve the problem are presented and discussed in detail. In this chapter the computational experiments performed to test the effectiveness of the heuristic algorithm and the branch and bound algorithm are presented. The complete experimental setup is discussed in this chapter and summary and conclusion of the results obtained is discussed in the next chapter.

7.1 Data Generation

In this work the release times and processing times and duedate are generated from the random number generator. Processing times were drawn from uniform integer distributions, the corresponding duedates were drawn from uniform distribution whose ranges are computed as a function of the sum of the respective processing times and release times.

7.2 Experimental Settings

The heuristic algorithm is tested for eight different sizes of n =5, 10, 15, 20, 25, 30, 35, 40 and same is for the branch and bound algorithm. The release times, processing times each job are drawn independently from uniform distributions

p_j is randomly generated in the interval [1,100],
r_j is randomly generated in the interval [0,k_1n],
d_j is randomly generated in the interval [r_j+p_j,$r_j+p_j+k_2$n]
k_1=20, k_2=1
The numbers of tardy jobs of the B&B and heuristic algorithms are recorded.

7.3 Software Implementation

Both the heuristic and the B&B optimal algorithms are coded in Java-programming language (NetBeans IDE 6.1).

The codes are run on a Pentium 4 CPU – 3.02 GHz, 512MB RAM computer in Windows environment. The number of tardy jobs in the schedules using the branch and bound and heuristic algorithms are obtained.

7.4. Analysis and Discussion of the Computational Results

This section presents the results of the computational experiments performed and observed effectiveness of the heuristic algorithm and the B&B algorithm. Both algorithms were tested on same data sets of release times, processing times and duedates. The computational results obtained are presented in Tables 7.1.

No of jobs	Tardy jobs (Heuristic Algorithm)	Tardy jobs (B&B)
5	0-2	0-2
10	1-5	1-5
15	4-8	3-7
20	9-14	8-13
25	13-17	12-16
30	14-22	13-21
35	16-25	14-23
40	22-29	20-27

Table 7.1: Result of Heuristic and B& B

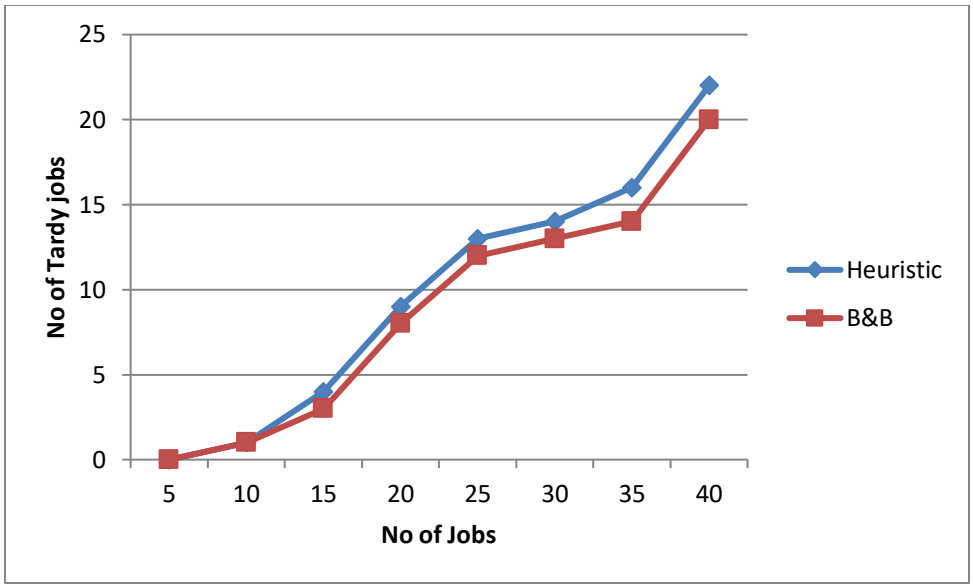

Figure: 7.1 Number of tardy jobs versus number of jobs (number of instances=50, minimum tardy jobs)

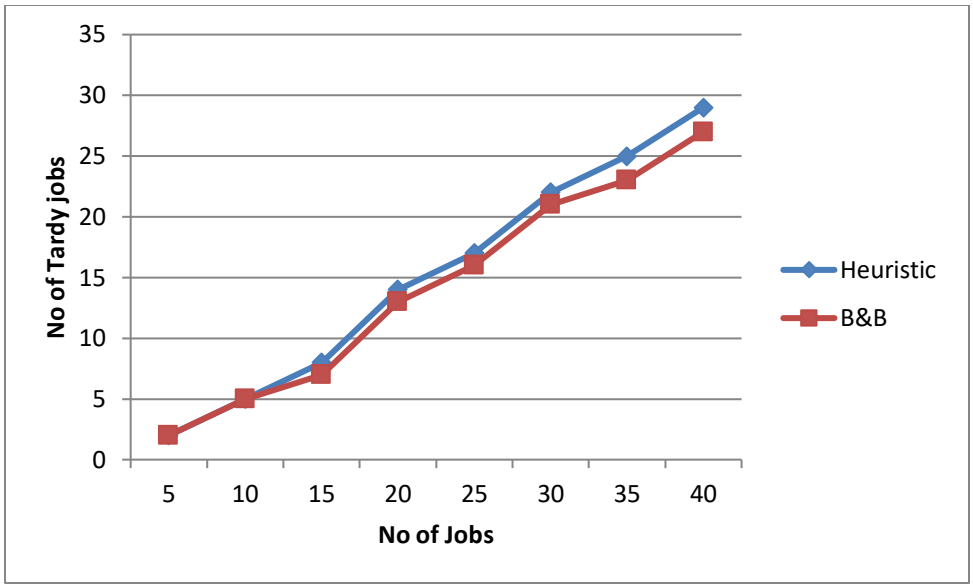

Figure 7.2: Number of tardy jobs versus number of jobs (number of instances =50, maximum tardy jobs)

From the above results it can be concluded that the heuristic algorithm gives the solution which is near to branch and bound. For the less number of jobs both the approaches give the same solution, however as the number of jobs increases, the branch and bound approach gives less number of tardy jobs.

CHAPTER 8

CONCLUSION AND FUTURE RESEARCH

In Chapter 7 the experimental setup and results obtained are presented. In this chapter, summary and directions for future research are given.

8.1 Summary

In this thesis, we addressed problem related to minimize the number of tardy jobs, a single machine scheduling problem with release time constraints performance objective of minimizing the number of tardy jobs.

The problem is proved to be NP-Hard problem, and only can be solved in pseudo polynomial time. We studied a heuristic algorithm and the branch and bound algorithm to solve the problem, implemented both and compared the results of both the algorithms and proved that the heuristic algorithm gives solution very near to branch and bound procedure based on the master sequence.

It is found that the numbers of tardy jobs obtained from B&B are less as compared to heuristic when the number of jobs increases.

8.2 Conclusions

For scheduling problem to minimize the number of tardy jobs with release time constraints, from the implemented heuristic approach and branch and bound approach, following conclusions can be made

The heuristic algorithm gives a solution near to B&B for the $1|r_j|\Sigma U_j$. However as the number of the jobs increases the B&B gives less number of tardy jobs.

8.3 Suggestions for Future Research

The scheduling problem discussed in this thesis is deterministic single machine scheduling problem with a performance objective of minimizing the number of tardy jobs with release time constraints without preemption. As customization is the key feature of any product today, we need more study and research in this area, so that we can provide benchmarks for many of the application area of scheduling.

It would be interesting if the solution obtained by the implemented heuristic in this thesis is improved to more near optimal solution by using meta-heuristic approaches like tabu search, simulated annealing or genetic algorithm techniques and also more interesting if the case is non-deterministic.

CHAPTER 7

COMPUTATIONAL EXPERIMENTS AND RESULTS

In the last chapter the heuristic solution approach and the branch and bound procedure based on master sequence to solve the problem are presented and discussed in detail. In this chapter the computational experiments performed to test the effectiveness of the heuristic algorithm and the branch and bound algorithm are presented. The complete experimental setup is discussed in this chapter and summary and conclusion of the results obtained is discussed in the next chapter.

7.1 Data Generation

In this work the release times and processing times and duedate are generated from the random number generator. Processing times were drawn from uniform integer distributions, the corresponding duedates were drawn from uniform distribution whose ranges are computed as a function of the sum of the respective processing times and release times.

7.2 Experimental Settings

The heuristic algorithm is tested for eight different sizes of n =5, 10, 15, 20, 25, 30, 35, 40 and same is for the branch and bound algorithm. The release times, processing times each job are drawn independently from uniform distributions

p_j is randomly generated in the interval [1,100],
r_j is randomly generated in the interval [0,k_1n],
d_j is randomly generated in the interval [r_j+p_j,$r_j+p_j+k_2$n]
k_1=20, k_2=1
The numbers of tardy jobs of the B&B and heuristic algorithms are recorded.

7.3 Software Implementation

Both the heuristic and the B&B optimal algorithms are coded in Java-programming language (NetBeans IDE 6.1).

The codes are run on a Pentium 4 CPU – 3.02 GHz, 512MB RAM computer in Windows environment. The number of tardy jobs in the schedules using the branch and bound and heuristic algorithms are obtained.

7.4. Analysis and Discussion of the Computational Results

This section presents the results of the computational experiments performed and observed effectiveness of the heuristic algorithm and the B&B algorithm. Both algorithms were tested on same data sets of release times, processing times and duedates. The computational results obtained are presented in Tables 7.1.

No of jobs	Tardy jobs (Heuristic Algorithm)	Tardy jobs (B&B)
5	0-2	0-2
10	1-5	1-5
15	4-8	3-7
20	9-14	8-13
25	13-17	12-16
30	14-22	13-21
35	16-25	14-23
40	22-29	20-27

Table 7.1: Result of Heuristic and B& B

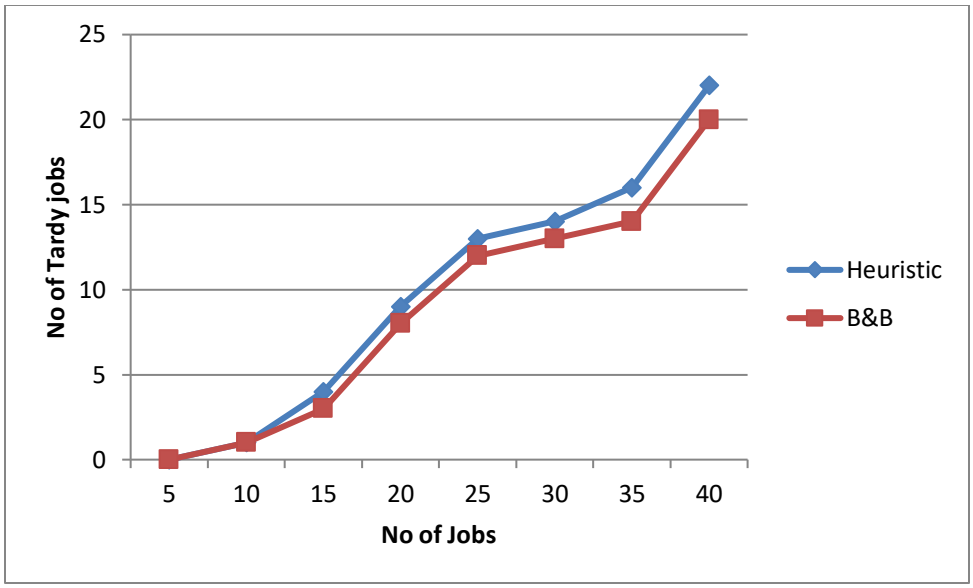

Figure: 7.1 Number of tardy jobs versus number of jobs (number of instances=50, minimum tardy jobs)

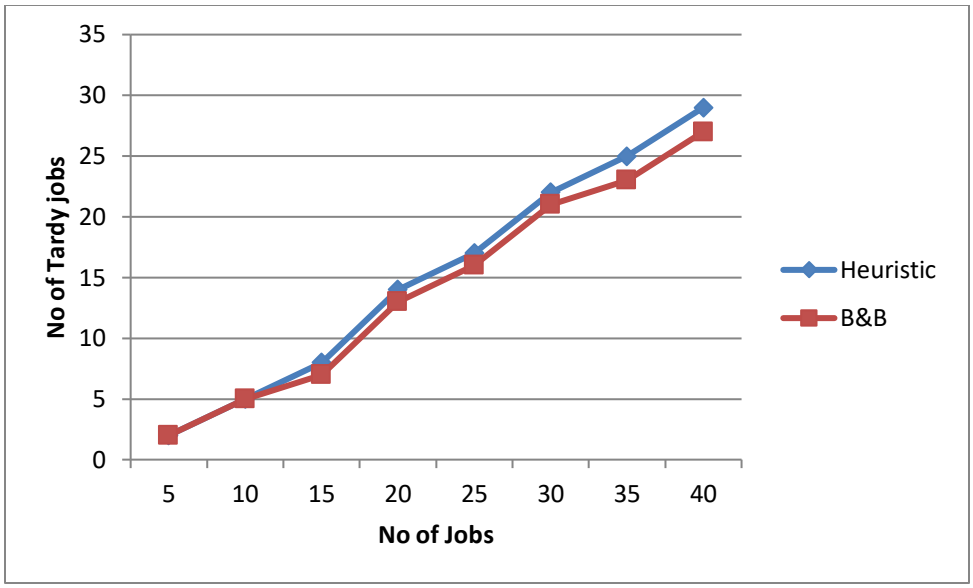

Figure 7.2: Number of tardy jobs versus number of jobs (number of instances =50, maximum tardy jobs)

From the above results it can be concluded that the heuristic algorithm gives the solution which is near to branch and bound. For the less number of jobs both the approaches give the same solution, however as the number of jobs increases, the branch and bound approach gives less number of tardy jobs.

CHAPTER 8

CONCLUSION AND FUTURE RESEARCH

In Chapter 7 the experimental setup and results obtained are presented. In this chapter, summary and directions for future research are given.

8.1 Summary

In this thesis, we addressed problem related to minimize the number of tardy jobs, a single machine scheduling problem with release time constraints performance objective of minimizing the number of tardy jobs.

The problem is proved to be NP-Hard problem, and only can be solved in pseudo polynomial time. We studied a heuristic algorithm and the branch and bound algorithm to solve the problem, implemented both and compared the results of both the algorithms and proved that the heuristic algorithm gives solution very near to branch and bound procedure based on the master sequence.

It is found that the numbers of tardy jobs obtained from B&B are less as compared to heuristic when the number of jobs increases.

8.2 Conclusions

For scheduling problem to minimize the number of tardy jobs with release time constraints, from the implemented heuristic approach and branch and bound approach, following conclusions can be made

The heuristic algorithm gives a solution near to B&B for the $1|r_j|\Sigma U_j$. However as the number of the jobs increases the B&B gives less number of tardy jobs.

8.3 Suggestions for Future Research

The scheduling problem discussed in this thesis is deterministic single machine scheduling problem with a performance objective of minimizing the number of tardy jobs with release time constraints without preemption. As customization is the key feature of any product today, we need more study and research in this area, so that we can provide benchmarks for many of the application area of scheduling.

It would be interesting if the solution obtained by the implemented heuristic in this thesis is improved to more near optimal solution by using meta-heuristic approaches like tabu search, simulated annealing or genetic algorithm techniques and also more interesting if the case is non-deterministic.

Appendix A
Program Code of Various Algorithms

We give the source code for the algorithms discussed in Chapter 6 and Chapter 5.The code are in java programming language (NetBeans IDE 6.1). Each java file contains a complete console based application. Below is a summary of all program files and their details.

filehandling.java

```
package thesis1.dao;
import java.io.*;
import java.util.*;
import thesis1.pojo.Address;
public class filehandling {
public static ArrayList<Address> handlefile( int no_of_jobs) throws IOException
   {
     RandomAccessFile fin=null;
     FileOutputStream fout=null;
     PrintWriter pw=null;
     Scanner in=new Scanner(System.in);
     ArrayList<Integer>duearr=new ArrayList<Integer>();
     ArrayList<Address>jobs=new ArrayList<Address>();
     int rel=0,pro=0,dued=0,k11=20,k2=1,n=no_of_jobs;
     fout=new FileOutputStream("randomnumbers.txt");
     pw=new PrintWriter(fout,true);
     pw.print("job processing release duedate");
    pw.println();
for(int rn=1;rn<=no_of_jobs;rn++)
{
```

```
 int rn1=rn;
 Random gen=new Random();
 pro=1+gen.nextInt(100);
 rel=k11*gen.nextInt(n);
 dued=(rel+pro)+(rel+pro+k2*gen.nextInt(n));
 if (duearr.isEmpty())
 duearr.add(dued); // add the number to the list
  else
     {
      while (!isAvailable(dued,duearr))
      dued =(rel+pro)+(rel+pro+k2*gen.nextInt(n));

       duearr.add(dued);
     }
  jobs.add(new Address(rn1,pro,rel,dued));
 pw.print(Integer.toString(rn1));
 pw.print("\t"+Integer.toString( pro));
 pw.print("\t"+Integer.toString(rel));
 pw.print("\t"+Integer.toString(dued));
 System.out.println("job is"+rn1+"processing is"+pro+"release is"+rel+"duedate is "+dued);
 pw.println();
}
return jobs;
   }
static  boolean isAvailable(int check,ArrayList<Integer> arr)
        {
          for (int x = 0; x < arr.size(); x++)
           if (check == arr.get(x).intValue())
            return false;
            return true;
        }
public static void main(String args[])throws IOException
{
   Scanner in=new Scanner(System.in);
   System.out.println("enter the number of jobs");
   int no_of_jobs=in.nextInt();
   filehandling f1=new filehandling();
   ArrayList<Address> f2= f1.handlefile(no_of_jobs);
 } }
```

Address.java

```
package thesis1.pojo;
 public class Address {
 public  int id=0;
 public  int processing=0;
 public int release=0;
 public int duedate=0;
```

```
 public Address(int id1,int p,int r,int d)
 {
  id=id1;  processing=p;  release=r;  duedate=d;
 }
@ Override
 public String toString()
 {
 return id+"\t"+processing+"\t"+release+"\t"+duedate;
 } }
```

Hrtest1.java

```
package thesis1.dao;
import java.io.*;
import java.util.*;
import java.util.Collections;
import thesis1.pojo.Address;
public class Hrtest1
{
   static LinkedList<Integer> re=new LinkedList<Integer>();
   static ArrayList<Integer> fi=new ArrayList<Integer>();
   static LinkedList<Address> o=new LinkedList<Address>();
   static int co=0,t=0,sizeo=0;
   public static void main(String args[])throws IOException
      {
        Cputime c=new Cputime();
        long start=0 ,end=0;
        long startcpu=0 ,endcpu=0;
        startcpu=c.getCpuTime();
        start=System.currentTimeMillis();
        LinkedList<Address> a1=new LinkedList<Address>();
        ArrayList<Integer>duearr=new ArrayList<Integer>();
        int rel=0,pro=0,dued=0,k11=20,k2=1;
        String h=null,b1=null;
        int k=0,j=0,i=0;
        LinkedList<Address> Job=new LinkedList<Address>();
        LinkedList<Integer> dates=new LinkedList<Integer>();
        RandomAccessFile  fin=new RandomAccessFile("randomnumbers.txt","r");
        h=fin.readLine();
         do
           {
            if(b1!=null)
             {
               System.out.println();
               while(Character.isDigit(b1.charAt(i)))
                   {  i++; }
                String jobid=b1.substring(0,i);
               int job_id=Integer.parseInt(jobid);
                k=2;
```

```
                while(!Character.isDigit(b1.charAt(k)))
                    {  k++;   }
                j=k;
                while(Character.isDigit(b1.charAt(j)))
                   {  j++; }
                String proce=b1.substring(k,j);
                int processing_time=Integer.parseInt(proce);
                k=j;
                while(!Character.isDigit(b1.charAt(k)))
                  { k++; }
                j=k;
                while(Character.isDigit(b1.charAt(j)))
                 {   j++;   }
                String relea=b1.substring(k,j);
                int release_time=Integer.parseInt(relea);
                k=j;
                while(!Character.isDigit(b1.charAt(k)))
                  {  k++;  }
String dueda=b1.substring(k,b1.length()).trim();
int due_date=Integer.parseInt(dueda);
a1.add(new Address(job_id,processing_time,release_time,due_date));
}}
  while((b1=fin.readLine())!=null);
fin.close();
LinkedList<Address>get_jobs=getjob(a1);
end=System.currentTimeMillis();
endcpu=c.getCpuTime()-startcpu;
System.out.println("elapsed time is" +(end-start)/1000+"secs");
System.out.println("cpu time is "+endcpu/100000000);
   }
public static  LinkedList<Address>  getjob(LinkedList<Address> a1)
{
 int i=0,releasemin=0,maxtime=0,removejob=0,minprocessing=0,j1=0,sj=0,k=0,nowfk=0;
 int ll=0;
 int jl=0;
 int mindue=0,fj=0,completiontime=0,t1=0,ret=0,minrelease=0;
 int incr=0,findmincomp=0,job=0;
 LinkedList<Address> f=new LinkedList<Address>();
 LinkedList<Address> s=new LinkedList<Address>();
 LinkedList<Address> tempvalueo=new LinkedList<Address>();
 LinkedList<Address> a2=new LinkedList<Address>();
 LinkedList<Address> L=new LinkedList<Address>();
 LinkedList<Integer> pr=new LinkedList<Integer>();
 LinkedList<Integer> du=new LinkedList<Integer>();
 ArrayList<Integer>nfk=new ArrayList<Integer>();
 ArrayList<Integer>checkmax=new ArrayList<Integer>();
 ArrayList<Integer>re1=new ArrayList<Integer>();
 ArrayList<Integer> findmax=new ArrayList<Integer>();
 ArrayList<Integer> mincompletion=new ArrayList<Integer>();
```

```
ArrayList<Integer> jobfind=new ArrayList<Integer>();
ArrayList<Integer> processinglinked=new ArrayList<Integer>();
   f=a1;
   a2.add(a1.get(1));
   for (int r=0;r<f.size();r++)
    {
     pr.add(a1.get(r).processing);
     re.add(a1.get(r).release);
     du.add(a1.get(r).duedate);
    }
   pr.clear();
   du.clear();
   re.clear();
   for(int rr=0;rr<f.size();rr++)
     {   re.add(f.get(rr).release);  }

  t=Collections.min(re);
  re.clear();
  while(f.size()>0)//start of top while
   {
    for(int fk=0;fk<f.size();fk++)
    {
     if(f.get(fk).release<=t)
      {
       nfk.add(fk);
       du.add(f.get(fk).duedate);          }       }
        mindue=Collections.min(du);
         du.clear();
    for(int fk=0;fk<f.size();fk++)   //start of for first
      {
       if(mindue==f.get(fk).duedate) //start of for first
         {
          o.add(f.get(fk));
          fj=fk;
          sj=t;
          co=t+f.get(fk).processing;
       if(co>f.get(fk).duedate)//start of first ifs
        {
        for(incr=0;incr<o.size();incr++)
         {
           for( i=0;i<o.size();i++)
            {               s.add(o.get(i));     }
       if(s.size()>1)
        {   s.remove(incr);          }
       for(  k=0;k<s.size();k++)
        {            re1.add(s.get(k).release);    }
       t1=Collections.min(re1);
       re1.clear();
try
```

```
  {
   for(int j=0;j<s.size();j++)
   {
    sizeo=s.size();
    completiontime=s.get(j).processing+t1;
    if(j<sizeo-1)
     {
      findmax.clear();
      findmax.add(completiontime);
      findmax.add(s.get(j+1).release);
      t1=Collections.max(findmax);
}//end of if }//end of for }//end of try
catch(Exception e)
 { System.out.println("error"+e); }
s.clear();
mincompletion.add(completiontime);
job=incr;
jobfind.add(job);
}
findmincomp=Collections.min(mincompletion);
int c=0;
for( int k1=0;k1<mincompletion.size();k1++)
{
 if(mincompletion.get(k1)==findmincomp)
 {   ret= k1;  } }
mincompletion.clear();
L.add(o.get(ret));
o.remove(ret);
int sz=o.size();
for(int rr=0;rr<o.size();rr++)
 {    re.add(o.get(rr).release);  }
 t=Collections.min(re);
 re.clear();
 co=showcompletion(o);
  }//end of first if
 for(int lj=0;lj<L.size();lj++)//start of second if for
 {
  if((L.get(lj).release<=co))
  {
   if((co+L.get(lj).processing)<=L.get(lj).duedate)//start of second if
   {
    processinglinked.add(L.get(lj).processing);
   }//end of if1   }//end of if2  }//end of second if for
if(processinglinked.size()>0)
{
 minprocessing=Collections.min(processinglinked);
 for(int lj=0;lj<L.size();lj++)//start of third for
    {
     if(minprocessing==L.get(lj).processing)
```

```
    {
     jl=lj;       }//end of if     }//end of for
            if(L.get(jl).processing<o.get(o.size()-1).processing)
                {
                tempvalueo.clear();
                tempvalueo.add(o.get(o.size()-1));
                o.remove(o.size()-1);//removing last element in o
                o.add(L.get(jl));
                if(L.size()>0)
                 {  L.remove(jl);   }
                L.add(tempvalueo.get(0));
                for(int rr=0;rr<o.size();rr++)
                   {
                  re.add(o.get(rr).release);
                    }
               t=Collections.min(re);
               re.clear();
               co=showcompletion(o);
               try
                  {
               for(int rr1=0;rr1<o.size();rr1++)//start of for 1
                {
                  sizeo=o.size();
                  co=t+o.get(rr1).processing;
                  if(rr1<sizeo-1)
                  {
                    fi.clear();
                    fi.add(co);
                    fi.add(o.get(rr1+1).release);
                    t=Collections.max(fi);
                    fi.clear();
                   }//end of if  }//end of for 1  }//end of try
   catch(Exception e)
          {   System.out.println("error"+e);   }
                 }//end of if
}
processinglinked.clear();
}//end of if first
                  } //end of for first
  f.remove(fj);
for(int rr=0;rr<f.size();rr++)
{ re.add(f.get(rr).release); }
if(re.size()>0)
{ releasemin=Collections.min(re); }
checkmax.add(releasemin);
checkmax.add(co);
t=Collections .max(checkmax);
     re.clear();
checkmax.clear();
```

```
}//end of top while
System.out.println("\nsize of  L is "+L.size());
System.out.println("size of o is "+o.size());
return o;
}
static  int showcompletion(LinkedList<Address> o)
{
 try
                    {
                  for(int rr1=0;rr1<o.size();rr1++)//start of for 1
                     {
                      sizeo=o.size();
                       co=t+o.get(rr1).processing;
                     if(rr1<sizeo-1)
                        {
                      fi.clear();
                      fi.add(co);
                      fi.add(o.get(rr1+1).release);
                     t=Collections.max(fi);
                      fi.clear();
              }//end of if   }//end of for 1  }//end of try
    catch(Exception e)
        {    System.out.println("error"+e);          }
return co;
  }  }//end of class
```

Relaxationrelease.java

```
package thesis1.dao;
import thesis1.pojo.Address;
import java.util.Collections;
import java.util.LinkedList;
public class Relaxationrelease
{
static LinkedList<Address>putlinked=new LinkedList<Address>();
static LinkedList<Address> Jbar=new LinkedList<Address>();
static LinkedList<Integer> re=new LinkedList<Integer>();
static int lastelement=0, min1=0,jobtoadd=0,lastposition=0,incr=0,removemin=0;
static int dmax=0,pos=0,jobtoremove=0;
static LinkedList<Address>Ordered=new LinkedList<Address>();
 public  void sortduedate(LinkedList<Address>Job)
{
 LinkedList<Address>Ordered=new LinkedList<Address>();
 LinkedList<Integer> re=new LinkedList<Integer>();
 int pos=0,jobtoremove=0;
 Ordered.clear();
 for(int i=0;i<Job.size();i++)
 {   Ordered.add(Job.get(i));  }
```

```
 Job.clear();
int sz=Ordered.size();
for(int i=0;i<sz;i++)
{
for(int j=0;j<Ordered.size();j++)
{ re.add(Ordered.get(j).duedate);}
min1=Collections.min(re);
re.clear();
for(int k=0;k<Ordered.size();k++)
{
if(min1==Ordered.get(k).duedate)
{ jobtoremove=k; } }
Job.add(Ordered.get(jobtoremove));
Ordered.remove(jobtoremove);
} }
 public  void releaserelax(LinkedList<Address>temp)
{
int dmax=0;
dmax=temp.get(0).release;
for(int i=1;i<temp.size();i++)
{
 if(temp.get(i).release<dmax)
  {    temp.get(i).release=dmax;    }
  else
   {
    dmax=temp.get(i).release;
    }   } }}
```

Sortrelease.java

```
package thesis1.dao;
import thesis1.pojo.Address;
import java.util.*;
import java.util.Collections;
 public class sortrelease {
  static LinkedList<Address> Jbar=new LinkedList<Address>();
  static LinkedList<Address> Sigma=new LinkedList<Address>();
  static LinkedList<Integer> dates=new LinkedList<Integer>();
  static LinkedList<Integer> copydates=new LinkedList<Integer>();
  static LinkedList<Integer> releases=new LinkedList<Integer>();
    static LinkedList<Address> sortreleasedates(LinkedList<Address>Job)
    {
     LinkedList<Address> o=new LinkedList<Address>();
     LinkedList<Address> mainjob=new LinkedList<Address>();
     ArrayList<Integer>tor=new ArrayList<Integer>();
     while(Job.size()>0)
      {
     for(int r=0;r<Job.size();r++)
      {       releases.add(Job.get(r).release)   ;       }
```

```
    int minr=Collections.min(releases);
   releases.clear();
    for(int j=0;j<Job.size();j++)
    {       o.add(Job.get(j))   ;      }
    for(int k=0;k<o.size();k++)
    {
     if(o.get(k).release==minr)
        {           tor.add(k)   ;          }
    }
    if(tor.size()==1)
    {
     int s=tor.get(0);
     mainjob.add(o.get(s));
     Job.remove(s);
    }
    else if(tor.size()>1)
    {
     int count=0;
     for(int rt=0;rt<tor.size();rt++)
     {
       int rs=tor.get(rt)  ;
       if(rt==0)
       {
        mainjob.add(o.get(rs));
        Job.remove(rs);
        count=count+1;
       }
       else{
           mainjob.add(o.get(rs));
           Job.remove(rs-count);
           count=count+1;
       }       }      }
    tor.clear();
    o.clear();
            } //end of while
    return mainjob;
 }}
```

Kissealgorithm1.java

```
package thesis1.dao;
import thesis1.pojo.Address;
import java.util.*;
class Kissealgorithm1
{
static int t=0,co=0,sizeo,m=0,l=0,csi=0,i=0,csi1=0,completiontime1=0;
static LinkedList<Integer> re=new LinkedList<Integer>();
static LinkedList<Integer> fi=new LinkedList<Integer>();
static LinkedList<Address> Si=new LinkedList<Address>();
```

```
static LinkedList<Address> returnjobs=new LinkedList<Address>();
static   LinkedList<Integer>findmax=new  LinkedList<Integer>();
static int sizejob=0;
static LinkedList<Address> Si1=new LinkedList<Address>();
static int   function(LinkedList<Address> Job)
{
 sizejob=Job.size();
 LinkedList<Address> E1=new LinkedList<Address>();
 LinkedList<Address> E=new LinkedList<Address>();
 LinkedList<Address> temp=new LinkedList<Address>();
 for(int rr=0;rr<Job.size();rr++)
     {
      re.add(Job.get(rr).release);
      }
 t=Collections.min(re);
 re.clear();
 try
   {
    for(int rr1=0;rr1<Job.size();rr1++)//start of for 1
      {
       E1.add(Job.get(rr1));
       sizeo=Job.size();
       co=showcompletion(E1);
       if(co<=Job.get(rr1).duedate)
         {
           Si.clear();
           Si1.clear();
           E.add(Job.get(rr1));
            }//end of if
        else
           {
            int getl=computel(rr1,E1);
            E.add(Job.get(rr1));
            E.remove(getl);
            E1.remove(getl);
            co=showcompletion(E);
            }//end of  else   }//end of for 1 }//end of try
  catch(Exception e)
     {        System.out.println("error"+e);       }
   return E.size();
}
static int computel(int rr1,LinkedList<Address>Job)
{
 int siz=Job.size();
 try
  {
   m=rr1;
   if(m==0)
     {
```

```
    l=m;
   }//end of if inside else
  else
    {
     Si.clear();
     Si1.clear();
     i=1;
     Si.add(Job.get(i));
     csi=Si.get(i-1).release+Si.get(i-1).processing;
     Si1.add(Job.get(i-1));
     csi1=Si1.get(i-1).release+Si1.get(i-1).processing;
     l=0;
     while(i<Job.size())
          {
           if(csi>csi1)
             {
              try
              {
               Si.clear();
              if(Si1.size()>0)
                   {
                    for(int ad=0;ad<Si1.size();ad++)
                       {
                       Si.add(Si1.get(ad));
                        }    }    }//end of try

        catch(Exception e)
           {              System.out.println("error"+e);              }
        csi=csi1;
        l=i;
        }//end of first if inside the while
       if(i<m)
        {
         if(i<siz-1)
          {    Si.add(Job.get(i+1));  }
         csi=showcompletion(Si);
         Si1.add(Job.get(i));
         csi1=showcompletion(Si1);
           }
         i=i+1;
       }//end of while        }//end of inner else
    }
  catch(Exception e)
  {     System.out.println("error"+e);    }
return l;
}
static int showcompletion(LinkedList<Address> E)
{
for(int rr=0;rr<E.size();rr++)
```

```
   {
    re.add(E.get(rr).release);
   }
  t=Collections.min(re);
  re.clear();
  try
   {
    for(int rr1=0;rr1<E.size();rr1++)//start of for 1
          {
           sizeo=E.size();
           co=t+E.get(rr1).processing;
           if(rr1<sizeo-1)//j+1 at last causes an error
             {
              fi.clear();
              fi.add(co);
              fi.add(E.get(rr1+1).release);
              t=Collections.max(fi);
               }//end of if
           }//end of for 1
        }//end of try

  catch(Exception e)
      {
      System.out.println("error"+e);
      }
 return co;
}//end of function
}//end of main class
```

Mastersequence.java

```
package thesis1.dao;
import thesis1.pojo.Address;
import java.util.*;
import java.util.Collections;
public class Mastersequence
{
  static LinkedList<Address> Sigma1=new LinkedList<Address>();
    public LinkedList<Address> master(LinkedList<Address> Job)
     {
    LinkedList<Address> Jbar=new LinkedList<Address>();
    LinkedList<Address> Sigma=new LinkedList<Address>();
    LinkedList<Integer> dates=new LinkedList<Integer>();
    LinkedList<Integer> copydates=new LinkedList<Integer>();
    int  min1=0,jobtoadd=0,lastposition=0,incr=0,removemin=0;
        for(int k=0;k<Job.size();k++)
      {
          Sigma.add(Job.get(k));
          Jbar.add(Job.get(k));
```

```
        if(Jbar.size()>0)
        {
          lastposition=Jbar.size()-1;
          for(int j=0;j<Jbar.size();j++)//for 1
          {
           if(Jbar.get(j).duedate>Jbar.get(lastposition).duedate)
                   {
                  dates.add(Jbar.get(j).duedate);
                     }
                 }
              if(dates.size()==1)
              {
                  for(int jd=0;jd<Job.size();jd++)
                    {
                     if(dates.get(0)==Job.get(jd).duedate)
                        {
                         jobtoadd=jd;
                         }    }
                   Sigma.add(Job.get(jobtoadd));
                 }
               else if(dates.size()>1)
              {
                    for(int  i=0;i<dates.size();i++)
                          {  copydates.add(dates.get(i));   }
                for(int tk=0;tk<dates.size();tk++)
                {
                 min1=Collections.min(copydates);
                 for(int jd=0;jd<Job.size();jd++)
                    {
                     if(min1==Job.get(jd).duedate)
                       {  jobtoadd=jd;    }    }
                      Sigma.add(Job.get(jobtoadd));
                       for(int cd=0;cd<copydates.size();cd++)
                       {
                           if(min1==copydates.get(cd))
                         {
                         removemin=cd;
                            }}
                              copydates.remove(removemin);
          incr=incr+1;
         } }
        dates.clear();
} }
return Sigma;
}}
```

Binarymain.java

```
package thesis1.dao;
```

```
import thesis1.pojo.Address;
import java.io.*;
import java.util.*;
import java.util.Collections;
class node
{
LinkedList<Address>value;
node leftc=null;
node rightc=null;
}
public class Binarymain
{
static LinkedList<Address>putdata1=new LinkedList<Address>();
static LinkedList<Address>putlinked=new LinkedList<Address>();
static LinkedList<Address>putlinked1=new LinkedList<Address>();
static LinkedList<Address>putlinked7=new LinkedList<Address>();
static LinkedList<Address>putlinked8=new LinkedList<Address>();
static LinkedList<Address>putlinked2=new LinkedList<Address>();
static LinkedList<Address>putlinked6=new LinkedList<Address>();
static LinkedList<Address>putlinked4=new LinkedList<Address>();
static LinkedList<Address>putlinked5=new LinkedList<Address>();
static LinkedList<Address>putlinked9=new LinkedList<Address>();
static LinkedList<Address>putlinked10=new LinkedList<Address>();
static LinkedList<Integer> re=new LinkedList<Integer>();
static LinkedList<Integer> pu1=new LinkedList<Integer>();
static ArrayList<Integer> fi=new ArrayList<Integer>();
static ArrayList<Integer> minindex=new ArrayList<Integer>();
static LinkedList<Address> putdata=new LinkedList<Address>();
static LinkedList<Address> getdata=new LinkedList<Address>();
static LinkedList<Address> leftl=new LinkedList<Address>();
static LinkedList<Address> temp=new LinkedList<Address>();
static LinkedList<Address> Sigmatemp=new LinkedList<Address>();
static LinkedList<Address> Sigmatemp1=new LinkedList<Address>();
static LinkedList<Address> lefttemp=new LinkedList<Address>();
static LinkedList<Address> rightl=new LinkedList<Address>();
static LinkedList<Address> righttemp=new LinkedList<Address>();
static LinkedList<Address> Sigmaroot=new LinkedList<Address>();
static ArrayList<Integer> lastpos=new ArrayList<Integer>();
static ArrayList<Integer> rightpos=new ArrayList<Integer>();
static ArrayList<Integer> relax=new ArrayList<Integer>();
static LinkedList<Address> Sigma1=new LinkedList<Address>();
static LinkedList<Address> Sigma2=new LinkedList<Address>();
static LinkedList<Address> Sigmaright=new LinkedList<Address>();
static LinkedList<Address> relaxtemp=new LinkedList<Address>();
static LinkedList<Address>newtemp=new LinkedList<Address>();
static LinkedList<Address>newtemp1=new LinkedList<Address>();
static LinkedList<Address>leftadded=new LinkedList<Address>();
static LinkedList<Address>leftadded1=new LinkedList<Address>();
static ArrayList<Integer> releasebefore=new ArrayList<Integer>();
```

```
static ArrayList<Integer> sizedata=new ArrayList<Integer>();
static ArrayList<Integer> leftpos=new ArrayList<Integer>();
static ArrayList<Integer> leftpos1=new ArrayList<Integer>();
LinkedList<Address> Upper_Bound=new LinkedList<Address>();
static int UB=0,dp=0,dp1=0,position=0,b=0,righttempsize=0;
static int co=0,t=0,sizeo=0,LB=0,res=0,s2=0;
public int branch_and_bound()throws IOException
{
 Cputime c=new Cputime();
 int optimal=0;
 LinkedList<Address> firstnode=new LinkedList<Address>();
 LinkedList<Address> left=new LinkedList<Address>();
 LinkedList<Address> right=new LinkedList<Address>();
 LinkedList<Address> test1right=new LinkedList<Address>();
 LinkedList<Address> Sigma=new LinkedList<Address>();
 LinkedList<Address> Job=new LinkedList<Address>();
 LinkedList<Address> mainjob=new LinkedList<Address>();
 long start=0 ,end=0;
 long startcpu=0,endcpu=0;
 startcpu=c.getCpuTime();
 System.out.println("startcpu is is"+startcpu);
 long startuser   =c.getUserTime( );
 System.out.println("startuseris"+startuser);
 start=System.currentTimeMillis();
 System.out.println("start is "+start);
 int lastelement=0, min1=0,jobtoadd=0,lastposition=0,incr=0,
 removemin=0;
 Address root=null;
 node rootc =null;
 ArrayList<Integer>duearr=new ArrayList<Integer>();
 int rel=0,pro=0,dued=0,k11=20,k2=1;
 RandomAccessFile fin=null;
 String h=null;
 String b1=null;
 int k=0,j=0,i=0;
 try
 {
  fin=new RandomAccessFile("randomnumbers.txt","r");
  h=fin.readLine();
   do
    {if(b1!=null)
       {
        System.out.println();
          while(Character.isDigit(b1.charAt(i)))
             {i++;}
         String jobid=b1.substring(0,i);
         int job_id=Integer.parseInt(jobid);
         k=2;
         while(!Character.isDigit(b1.charAt(k)))
```

```
            {k++;}
          j=k;
          while(Character.isDigit(b1.charAt(j)))
              {j++;}
          String proce=b1.substring(k,j);
          int processing_time=Integer.parseInt(proce);
          k=j;
          while(!Character.isDigit(b1.charAt(k)))
             {k++;}
         j=k;
         while(Character.isDigit(b1.charAt(j)))
             {j++;}
         String relea=b1.substring(k,j);
         int release_time=Integer.parseInt(relea);
         k=j;
         while(!Character.isDigit(b1.charAt(k)))
           {k++; }
         String dueda=b1.substring(k,b1.length()).trim();
         int due_date=Integer.parseInt(dueda);
         Job.add(new Address(job_id,processing_time,release_time,due_date));
      } }
    while((b1=fin.readLine())!=null);
    fin.close();
 }
catch(IOException e)
{System.out.println("error"+e);}
 sortrelease sr=new sortrelease();
 System.out.println("job is"+Job);
 mainjob=sr.sortreleasedates(Job);
 System.out.println("mainjob is"+mainjob);
 Mastersequence m1=new Mastersequence();
 Hrtest1 hrtest1=new Hrtest1();
 Sigma =m1.master(mainjob);
 Upper_Bound=hrtest1.getjob(mainjob);
 UB=Upper_Bound.size();
 Sigmaroot.add(Sigma.get(0));
 rootcalc(rootc,Sigmaroot,Sigma);
 while(righttemp.size()!=0)
  {
   int endpos=lastpos.get(b);
   firstnode=sublist1(Sigmatemp1,0,endpos);
   b=b+1;
   int endrightpos=rightpos.get(0);
   test1right=sublist5(righttemp,0,endrightpos);
   rightpos.remove(0);
   rootcalc(rootc,firstnode,test1right);
   Sigmatemp1.subList(0,endpos).clear();
   righttemp.subList(0,endrightpos).clear();
  }
```

```
if(sizedata.size()>0)
 {  optimal=Collections.max(sizedata);}
end=System.currentTimeMillis();
System.out.println("systemelapsed time is" +(end-start)/1000+"secs");
long enduser=c.getUserTime();
System.out.println("userelapsed time is" +(enduser-startuser)/1000000000+"secs");
endcpu=c.getCpuTime();
System.out.println("cpuelapsed time is" +(endcpu-startcpu)/1000000000+"secs");
return optimal;
}//end of branch_and_bound
public static void rootcalc(node rootc,LinkedList<Address>Sigmaroot,LinkedList<Address> Sigma)
{
  node tempd=new node();
  node tempc=new node();
  int ssize=Sigma.size();
  for(int sg=0;sg<Sigma.size()+ssize;sg++)
    {try {
         if(rootc==null)
          {
           tempd.value=Sigmaroot;
           tempd.leftc=null;
           tempd.rightc=null;
           rootc=tempd;
           }
         else
         {
          tempc=rootc;
          Sigmatemp=tempc.value;
          for(int sd=0;sd<Sigma.size();sd++)
            {
             if(Sigmatemp.getLast()==Sigma.get(sd))
                {
                 minindex.add(sd);
                }
            }
          if(minindex.size()>0)
            {
            dp=Collections.min(minindex);
            }
          if(minindex.size()>0)
            {
             lefttemp=sublist2(Sigma,dp+1,Sigma.size());
            }
          else if(minindex.size()<1)
           {
            lefttemp=Sigma;
           }
         minindex.clear();
         getdata.clear();
```

```
getdata=getfrom(rootc.value,lefttemp);
newtemp1.clear();
if(lefttemp.size()>0)
  {
   for(int i1=0;i1<Sigmaroot.size();i1++)
     {
      newtemp1.add(Sigmaroot.get(i1));
      }
   newtemp.clear();
   relaxtemp.clear();
   relax.clear();
   releasebefore.clear();
   for(int c1=0;c1<lefttemp.size();c1++)
     {
       newtemp.add(lefttemp.get(c1));
       relax.add(lefttemp.get(c1).duedate);
     }
  int kremove=0;
  int z=lefttemp.size();
  for(int x=0;x<z;x++)
     {
      for(int y=x+1;y<newtemp.size();y++)
       {
        if(newtemp.get(x).id==newtemp.get(y).id)
           {
            kremove=y;
            relax.remove(kremove);
           newtemp.remove(kremove);
       }} }
  int present=0;
  Relaxationrelease ra=new Relaxationrelease();
  Kissealgorithm1 ki=new Kissealgorithm1();
  ra.sortduedate(newtemp);
  for(int k=0;k<newtemp.size();k++)
     {releasebefore.add(newtemp.get(k).release);}
  ra.releaserelax(newtemp);
  for(int j3=0;j3<newtemp.size();j3++)
     {
      for(int h2=0;h2<newtemp1.size();h2++)
        {
         if((newtemp.get(j3).id)==(newtemp1.get(h2).id))
           {
            present=present+1;
            break;
            }
         else
           {present=0;}
       }
      if(present==0)
```

```
                {newtemp1.add(newtemp.get(j3));}
             }
          LB=ki.function(newtemp);
          for(int j4=0;j4<newtemp.size();j4++)
            {
             newtemp.get(j4).release=releasebefore.get(j4);
           }}
while(!getdata.isEmpty())
{
  if(getdata.size()>0)
  {
   if(getdata.size()==1)
   {
    relaxtemp.clear();
    for(int s=0;s<Sigmatemp.size();s++)
       {
        relaxtemp.add(Sigmatemp.get(s));
       }
   relaxtemp.add(getdata.get(0));
   leftpos.add(leftadded.size());
   if(leftadded.size()>0&&leftpos.size()>1)
       {
        res=compare(relaxtemp,leftadded,leftpos);
       }
  if(res==0)
      {
       if(tempc.leftc==null)
        {
         Sigma1=tempc.value;
         Sigma1.add(getdata.get(0));
         sizedata.add(Sigma1.size());
         leftpos.add(leftadded.size());
         for(int s=0;s<Sigma1.size();s++)
         {
          leftadded.add(Sigma1.get(s));
         }
        node tmp1=new node();
        tmp1.value=Sigma1;
        tmp1.leftc=null;
        tmp1.rightc=null;
        tempc.leftc=tmp1;
        tempc=tmp1;

              }
       else
        {
          rootc=rootc.leftc;
     } }}
  else if(getdata.size()>1)
```

```
{
 relaxtemp.clear();
 for(int s=0;s<Sigmatemp.size();s++)
  {
   relaxtemp.add(Sigmatemp.get(s));
  }
 relaxtemp.add(getdata.get(0));
 leftpos.add(leftadded.size());
 if(leftadded.size()>0&&leftpos.size()>1)
  {
   res=compare(relaxtemp,leftadded,leftpos);
   }
 int s3=0;
 if(res!=0)
    {Sigma.remove(0);}
 if(res==0)
   {
    if(tempc.leftc==null)
    {
     Sigma1=tempc.value;
     int firstpos=Sigma2.size();
     for(int s1=0;s1<Sigma1.size();s1++)
      {
       Sigma2.add(Sigma1.get(s1));
      }
     Sigma1.add(getdata.get(0));
     for(int s=0;s<Sigma1.size();s++)
      {
      leftadded.add(Sigma1.get(s));
       }
      sizedata.add(Sigma1.size());
      node tmp1=new node();
      tmp1.value=Sigma1;
      tmp1.leftc=null;
      tmp1.rightc=null;
      tempc.leftc=tmp1;
      tempc=tmp1;

 if(tempc.rightc==null)
  {
   rightl=sublist(Sigma2,firstpos,Sigma2.size());
   node tmp2=new node();
   tmp2.value=rightl;
   tmp2.leftc=null;
   tmp2.rightc=null;
   tempc.rightc=tmp2;
   tempc=tmp2;
   position=rightl.size();
   for(int sz=0;sz<rightl.size();sz++)
```

```
      {
       Sigmatemp1.add(rightl.get(sz));
      }
    lastpos.add(position);
    Sigmaright=sublist6(lefttemp,1,lefttemp.size());
    righttempsize=Sigmaright.size();
    rightpos.add(righttempsize);
    for(int m=0;m<Sigmaright.size();m++)
     {
      righttemp.add(Sigmaright.get(m));
     }    } }//end of left equal null
else
 {
  rootc=rootc.leftc;
 }} }}
getdata.clear();
 }}}
catch(Exception e)
{System.out.println("error"+e);}
  } }
public static int showcompletion(LinkedList<Address> left)
  {
   int co1=0,t1=0,sizel=0;
   ArrayList<Integer> fi1=new ArrayList<Integer>();
   try
    {
     for(int rr1=0;rr1<left.size();rr1++)
      {re.add(left.get(rr1).release);}
     t1=Collections.min(re);
     re.clear();
     for(int rr1=0;rr1<left.size();rr1++)//start of for 1
        {
          sizel=left.size();
          co1=t1+left.get(rr1).processing;
          if(rr1<sizel-1)
          {
           fi1.clear();
           fi1.add(co1);
           fi1.add(left.get(rr1+1).release);
           t1=Collections.max(fi1);
           fi1.clear();
          }}}//end of try
   catch(Exception e)
    {System.out.println("error"+e);}
return co1;
}
static LinkedList<Address> getfrom(LinkedList<Address> left,LinkedList<Address> Sigma)
 {
  int count=0;
```

```
  for(int i=0;i<left.size();i++)
   {
    temp.add(left.get(i));
   }
  try
   {
    for(int i=0;i<Sigma.size();i++)
    {
     for(int j=0;j<temp.size();j++)
     {
      if(Sigma.get(i).id==temp.get(j).id)
      {
       count=count+1;
       break;
      }
     else{count=0;}
     }
     if(count<1)//count is checked
       {
        temp.add(Sigma.get(i));
        co=showcompletion(temp);
        if(co>temp.getLast().duedate)
         { temp.removeLast();}
        else
          {
           putdata.add(temp.getLast());
           temp.removeLast();
          }}}
int pu=0;
  for(int k=0;k<putdata.size();k++)
   {
    for(int l=k+1;l<putdata.size();l++)
    {
     if(putdata.get(k).id==putdata.get(l).id)
     {
      pu1.add(l);
      int ks=pu1.get(0);
      putdata.remove(ks);
      pu1.clear();
     }}}
pu1.clear();
}
catch(Exception e)
{   System.out.println("error"+e); }
  temp.clear();
  return putdata;
}
public static LinkedList<Address> sublist(LinkedList<Address>Job,int start,int end)
{
```

```
  putlinked.clear();
  for(int i=start;i<end;i++)
   { putlinked.add(Job.get(i));}
  return putlinked;
}
public static LinkedList<Address> sublist1(LinkedList<Address>Job,int start,int end)
{
 putlinked1.clear();
 for(int i=start;i<end;i++)
 {putlinked1.add(Job.get(i));}
return putlinked1;
}
public static LinkedList<Address> sublist2(LinkedList<Address>Job,int start,int end)
{
 putlinked2.clear();
 for(int i=start;i<end;i++)
  {putlinked2.add(Job.get(i));}
return putlinked2;
}
public static LinkedList<Address> sublist4(LinkedList<Address>Job,int start,int end)
{
 putlinked4.clear();
 for(int i=start;i<end;i++)
 {putlinked4.add(Job.get(i));}
return putlinked4;
}
public static LinkedList<Address> sublist5(LinkedList<Address>Job,int start,int end)
{
  putlinked5.clear();
  for(int i=start;i<end;i++)
  {putlinked5.add(Job.get(i));}
  return putlinked5;
}
public static LinkedList<Address> sublist6(LinkedList<Address>Job,int start,int end)
{
putlinked6.clear();
for(int i=start;i<end;i++)
 {putlinked6.add(Job.get(i));}
return putlinked6;
}
public static LinkedList<Address> sublist7(LinkedList<Address>Job,int start,int end)
{
 putlinked7.clear();
 for(int i=start;i<end;i++)
  {putlinked7.add(Job.get(i));}
 return putlinked7;
}
public static LinkedList<Address> sublist9(LinkedList<Address>Job,int start,int end)
{
```

```
 putlinked9.clear();
 for(int i=start;i<end;i++)
  { putlinked9.add(Job.get(i));}
 return putlinked9;
}
public static LinkedList<Address> sublist10(LinkedList<Address>Job,int start,int end)
{
 putlinked10.clear();
 for(int i=start;i<end;i++)
 {putlinked10.add(Job.get(i));}
return putlinked10;
}
public static LinkedList<Address> sublist8(LinkedList<Address>Job,int start,int end)
{
 putlinked8.clear();
 if(Job.size()>0)
 {
    for(int i=start;i<end;i++)
  { putlinked8.add(Job.get(i));}
 }
return putlinked8;
}
public static int  compare(LinkedList<Address>Sigma1,LinkedList<Address>leftadded,ArrayList<Integer>leftpos)
{
 ArrayList<Integer> result=new ArrayList<Integer>();
 int count1=0,exist=0;
 for(int a2=0;a2<leftpos.size();a2++)
 {
  int x=leftpos.get(a2);
  leftpos1.add(x);
 }
 for(int a=0;a<leftpos1.size();a++)
   {
    leftadded1=sublist7(leftadded,leftpos1.get(0),leftpos1.get(1));
    leftpos1.remove(0);
    count1=0;
    if(leftadded1.size()>0)
     {
      if(Sigma1.size()==leftadded1.size())
       {
        for(int a3=0;a3<Sigma1.size();a3++)
         {
          if(Sigma1.get(a3).id==leftadded1.get(a3).id)
            {
             count1=count1+1;
            }}}}
 if(count1==Sigma1.size())
  {
   exist=1;
```

```
   result.add(exist);
  }
 else
 {
  exist=0;
  result.add(exist);
 }}
int a5=Collections.max(result);
result.clear();
return a5;
}
public static void main(String args[])throws IOException
{
    Scanner in=new Scanner(System.in);
    rnumbers f1=new rnumbers();
    Binarymain binarymain=new Binarymain();
    int x=binarymain.branch_and_bound();
    System.out.println("optimal length is"+x);
}}
public class Cputime {
public long getCpuTime( ) {
    ThreadMXBean bean = ManagementFactory.getThreadMXBean( );
    return bean.isCurrentThreadCpuTimeSupported( ) ?
        bean.getCurrentThreadCpuTime( ) : 0L;
}
 public long getUserTime( ) {
    ThreadMXBean bean = ManagementFactory.getThreadMXBean( );
    return bean.isCurrentThreadCpuTimeSupported( ) ?
        bean.getCurrentThreadUserTime( ) : 0L;
} }
```

REFERENCES

[1] Baker, K. R. (1974): Introduction to sequencing and scheduling. New York: John Wiley & Sons.

[2] Balut, S. J. (1973): Scheduling to minimize the number of late jobs when set-up and processing times are uncertain. Management Science, 19, 1283–1288.

[3] Baptiste, P. (1999a): An O (n^4) algorithm for preemptive scheduling of a single machine to minimize the number of late jobs. Operations Research Letters, 24, 175–180.

[4] Baptiste, P. (1999b): Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. Journal of Scheduling, 2, 245–252.

[5] Baptiste, P.; C. Le Pape and L. P´eridy (1998): Global constraints for partial CSPs: a case-study of resource and due date constraints. In M. Maher and J. F. Puget, editors: Proceedings of the Fourth International Conference on Principles and Practice in Constraint Programming. Lecture Notes in Computer Science. Volume 1520, Berlin: Springer, 87–101.

[6] Baptiste, P.; L. P´eridy and ´E. Pinson (2003): A branch and bound to minimize the number of late jobs on a single machine with release time constraints. European Journal of Operational Research, 144, 1–11.

[7] Blazewicz, J. C.; K. H. Ecker, E. Pesch and J. Weglarz (1996): Scheduling computer and manufacturing processes. 1st edition. Berlin: Springer.

[8] Blum M., "A Single machine –independent theory of the complexity of recursive functions", journal of the ACM 14, 2(1967)322-336

[9] Bomberger, E. E. (1966): A dynamic programming approach to a lot size scheduling problem. Management Science, 12, 778–784.

[10] Briand, C.; S. Ourari and B. Bouzouia (2006): On the minimization of late jobs in single machine scheduling with nested execution intervals. In International Conference Service Systems and Service Management. hURL: www.laas.fr/~briand/ BriandOurariMin_SumUI.pdfi.

[11] Carlier, J. (1981): Probl´emes d'ordonnancement `a dur´ees ´egales. QUESTIO, 5, 219–228.

[12]Carlier,J.andChretienne,P.,
problems'ordonnancement:modelisation/complexite/algorithms", Masson,Paris(1988).

[13] Chrobak, M.; C. D¨urr, W. Jawor, L. Kowalik and M. Kurowski (2004): A note on scheduling equal-length jobs to maximize throughput. Technical report hURL: http://www.citebase.org/abstract?id=oai:arXiv.org:cs/ 0410046i.

[14] Colorni M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini and M. Trubian. Heuristics from Nature for Hard Combinatorial Optimization Problems *International Transactions in Operational Research,* V 3, Issue 1, p1-21 (1996).

[15] Dauz`ere-P´er`es, S. (1995): Minimizing late jobs in the general one machine scheduling problem. European Journal of Operational Research, 81, 134–142.

[16] Dauz`ere-P´er`es, S. (1997): An efficient formulation for minimizing the number of late jobs in single machine scheduling. In IEEE 6th International Conference on Emerging Technologies and Factory Automation Proceedings, EFTA '97. Los Angeles, USA, 442–445.

[17] Dauz`ere-P´er`es, S. and M. Sevaux (1998b): A branch and bound method to minimize the number of late jobs on a single machine. In 6th International Workshop on Project Management and Scheduling. Istanbul, Turkey hURL: http://web.univ-ubs. fr/lester/~sevaux/Publications/inp-dauzere-98b-slides.pdfi.

[18] Dauz`ere-P´er`es, S. and M. Sevaux (2004): An exact method to minimize the number of tardy jobs in single machine scheduling. Journal of Scheduling, 7, 405–420.

[19] Deutsch,D., "Quantum theory ,the Church-Turing principle and the universal quantum computer" , Proceedings of the Royal Society of London A,400:97.(1985).

[20] Dhamala, T.N., and Kubiak, W, (2005): A brief Survey of Just-In-Time sequencing for mixed model production .International Journal of Operations Research, Vol.2, pp38-47.

[21] Dhamala,T.N.,and Khadka ,S.R.(2007):Just –In –Time Sequencing for mixed model Production Systems Revisited ,Discrete Optimization, submitted.

[22] Du, J. and J. Y. T. Leung (1990): Minimizing total tardiness on one machine is NP-hard. Mathematics of Operations Research, 15, 483–495.

[23] Glover (1986), scheduling the production of two component jobs on a single machine, European Journal of Operational Research 120, p 250-259.

[24] Gupta, S. K. and J. Kyparisis (1987): Single machine scheduling research. International Journal of Management Science, 15, 207–227.

[25] Hartmanis,J.and Stearns,R.E,"On the computational complexity of algorithms", Transactions of the AMS117(1965)285-306

[26] Ibarra, O. H. and C. E. Kim (1978): Approximation algorithms for certain scheduling problems. Mathematics of Operations Research, 3, 197–204.

[27] Jackson, J. R. (1955): Scheduling a production line to minimize maximum tardiness. Los Angeles, USA: University of California – Research Report 43, Management Research Project.

[28] Jang, W. and C. M. Klein (2002): Minimizing the expected number of tardy jobs when processing times are normally distributed. Operations Research Letters, 30, 100–106.

[29] Jawor,W(2005):Three Dozen papers on Online Algorithm. ACM SIGACT News, March, Vol36, no 1.

[30] Karp, R. M. (1972): Scheduling a production line to minimize maximum tardiness. In R. E. Miller and J. W. Thatcher, editors: Complexity of Computer Computation. New York: Plenum Press, 85–103.

[31] Kise, H.; T. Ibaraki and H. Mine (1978): A solvable case of the one-machine scheduling problem with ready and due times. Operations Research, 26, 121–126.

[32] Kyparisis, George J., C. Koulamas (2006), "Flexible Flow Shop Scheduling with Uniform Parallel Machines," European Journal of Operational Research, 168.3: 985-997.

[33] Lasserre, J. B. and M. Queyranne (1992): Generic scheduling polyhedra and a new mixed integer formulation for single machine scheduling. In 2nd Conference on Integer Programming and Combinatorial Optimization. Pittsburgh, USA: Carnegie- Mellon University, 139–149.

[34] Lawler, E. L. (1976): Sequencing to minimize the weighted number of late jobs. RAIRO Recherche Operationnel, 10, 27–33.

[35] Lawler, E. L. (1977): A “pseudopolynomial” algorithm for sequencing jobs to minimize total tardiness. Annals of Discrete Mathematics, 1, 331–342.

[36] Lawler, E. L. (1982): A fully polynomial approximation scheme for the total tardiness problem. Operations Research Letters, 1, 207–208.

[37] Lawler, E. L. (1990): A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs. Annals of Operations Research, 26, 125–133.

[38] Lawler, E. L. (1994): Knapsack-like scheduling problems, the Moore-Hodgson algorithm and the “tower of sets” property. Mathematical and Computer Modelling, 20, 91–106.

[39] Lawler, E. L.; J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys (1993): Sequencing and scheduling: algorithms and complexity. In S. C. Graves; A. H. G. Rinnooy Kan and P. H. Zipkin, editors: Logistics of Production and Inventory. Handbooks in Operations Research and Management Science. Volume 4, Amsterdam et al.: North-Holland, 445–522.

[40] Lenstra, J. K.; A. H. G. Rinnooy Kan and P. Brucker (1977): Complexity of machine scheduling problems. Annals of Discrete Mathematics, 1, 343–362.

[41] Li ma,S.D., Panwalkar,S.S and Thongmee,S.(1997): A Single Machine Scheduling Problem with Common Due Window and Controllable Processing Times. Annals of Operation Research, vol.70, pp145-154.

[42] Maxwell, W. L. (1970): On sequencing n jobs on one machine to minimize the number of late jobs. Management Science, 16, 295–297.

[43] M'Hallah, R. and R. L. Bulfin (2007): Minimizing the weighted number of tardy jobs on a single machine with release dates. European Journal of Operational Research, 176, 727–744.

[44] Milenkovie, M.(1997):Operating Systems. Tata McGraw-Hill.

[45] Moore, J. M. (1968): A n job, one machine scheduling algorithm for minimizing the number of late jobs. Management Science, 15, 102–109.

[46] Pinedo, M.,"scheduling–theory, algorithms, and systems", Prentice Hall, Englewood Cliffs(1995).

[46] P Brucker, (1995) Scheduling Algorithms. Springer Visit http://www.mathematik.uni-osnabrueck.de

[47] Pinedo, M., "scheduling –theory, algorithms, and systems", Prentice Hall, Englewood Cliffs (1995).

[48] Sidney, J. B. (1973): An extension of Moore's due date algorithm. In S. E. Elmaghraby, editor: Symposium on the Theory of Scheduling and its Applications. Lecture Notes in Economics and Mathematical Systems. Volume 86, Berlin: Springer, 393–398.

[49] Smith, W. E. (1956): Various optimizers for single-state production. Naval Research Logistic Quarterly, 3, 59–66.

[50] Tanenbaum, (2004): Modern Operating Systems. Prentice–Hall of India Pvt. Ltd.