# CHAPTER 1

# INTRODUCTION

The scheduling theory concerns the optimum allocation of limited resources like money, tools, energy, machines, and manpower over time to perform a set of activities (operation) in some processes like computation and manufacturing.

The first scheduling algorithms were appeared in the mid fifties, while the area developed as its own specific field during the year 1960's with many industrial applications. The computer scientists in the year 1970's accelerated investigations of scheduling as a tool for improving the performance of computer systemsOne of the simplest ways to understand the scheduling problem is to find the answer for following question:

Given a number of jobs that have to be processed on a machine consisting of a certain number of processor, find for each job a processor assignment for a certain time interval so that the completion time of the last finishing job is minimal. This time is called makespan of the schedule and it is one of many objective functions one could be interested in minimizing or maximizing.

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.

Most of the difficulties of optimal solution processes in scheduling theory lie on the fact that an examination of the solution space in combinatorial optimization is very much time consuming even it is a finite set.

A definition is quoted by Carlier and Chretienne: *"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.",* Carlier and Chretienne [1].

Another definition put forward by Pinedo: *"Scheduling concerns the allocation of limited resources to task over time. It is decision-making process that has a goal the optimization of one or more objectives.",*Pinedo [2].

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. We consider two types of resources: renewable resource (which is available after use e.g. machine, file, processor, personnel etc.) and non renewable resources (which disappear after use e.g. money, raw materials etc.). There are two types of optimality criteria those relating to completion time and those relating to costs. In the category of completion time related criteria we find for example those which measure the completion time of whose schedule and those which measure the tardiness of jobs in relation to their due date. In the category of cost related criteria we may cite those which represent cost of machine use and those which represent cost allied to waiting time of operations before and/or after they are processed.

Now days, the scheduling theory has become the most desirable concept in the computing environment; the hardware configuration of computer such as processor management, memory management, I/O Management and other resource management, and software development structure such as modular programming, object oriented, aspect oriented concepts. Today's most

popular computer processing such as on-line processing, data communication over the network and Artificial Intelligence (AI) technology are also characterized by scheduling theory[3].

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.

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 weighted 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.

## 1.1 Machine Scheduling

The problem of scheduling arises in several areas like production management, computer networks, operating systems and many fields having resource constraints. Scheduling concerns the allocation of limited resources to tasks over time. The general scheduling problem can be formulated in this way: We are given m machines, $M_i$ i = 1,2,...,m and n number of jobs j, j = 1,2,…,n. Besides, there is an objective function, which gives the cost of scheduling. The problem is to assign the jobs an allocation of one or more time intervals on one or more machines, minimizing the total objective value. The terms 'machine' and 'job' are very general. For example machine can be a microprocessor, a water pump, or even office personnel. Similarly, jobs can be of any type. If the machine is a microprocessor, then job means a program.

## 1.2 Three field notation

Scheduling problems can be described by a three field notation $\alpha|\beta|\gamma$ where $\alpha$ describes the machine environment, $\beta$ describes job characteristics, and $\gamma$ describes the objective functions to

be minimized, Graham et al [4]. A field may contain more than one entry but may also be empty. Latter, it is described in Chapter 3. Our problem is a single machine scheduling to minimize weighted number of tardy jobs with release time constant is denoted by in three field notations as $1||\sum W_j U_j$

## 1.3 Significance of the problem

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

## 1.4 Notations and Definitions

Let n be the number of jobs to be processed with $j \epsilon \{1,\ldots\ldots..,n\}$ denoting a typical job. Alternatively, a job is denoted $J_i$ and the set of jobs to be processed is given by $\{J_1,\ldots\ldots....,J_n\}$.Let $i \epsilon \{1,\ldots\ldots\ldots\ldots,m\}$ be a certain machine of a set of m machines available. There are a terms related to scheduling problems which are used throughout this dissertation [5]

### 1.4.1 Machine

A machine is available to execute jobs and tasks. Different machine environments exist. Such as single machine and parallel machines. Generally we use in this dissertation as single machine scheduling.

### 1.4.2 Release Time/Date

The point in time when a job j is ready to be processed. It is denoted by $r_j$. It is also known as arrival or ready time/date.

### 1.4.3 Processing Time

It is defined as the length of time to process a job or task. It is denoted by $p_j$

### 1.4.4 Completion Time

The time at which a job is finished. It is denoted by $C_j$

### 1.4.5 Due Date

The point in time at which job should be completed. It is denoted by $d_j$.

### 1.4.6 Waiting Time

Length of time between the ready time of a job and beginning of processing of a job. It is denoted by W.

### 1.4.7 Weight (Priority)

A weight can be added to the jobs to express relative urgency or priority between them. It is denoted by $w_j$.

### 1.4.8 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.

### 1.4.9 Lateness

Difference between completion time and due date i.e. $L_j=C_j-d_j$ where $C_j$ is the completion time job j and $d_j$ is the due date of job j.

### 1.4.10 Tardiness

The tardiness of job j $T_j$ is defined as $T_j=\max(0,C_j-d_j)$ where $C_j$ is the completion time of job j and $d_j$ is the due date of job j.

### 1.4.11 Flow time

Amount of time job j spends in the system .$F_j=C_j-r_j$, where $C_j$ is the completion time of job j and $r_j$ is the release date of job j.

### 1.4.12 Earliness

Difference between the due date and the completion time i.e $E_j=d_j-C_j$, where $d_j$ is the due date of job j and $C_j$ is the completion time of job j.

### 1.4.13 Preemption

The preemption (or job splitting) is allow 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. In this case we consider only the non preemption.

### 1.4.14 Slack Time

Time until a jobs due date minus the processing time of a jobs.

### 1.4.15 Precedence

Some jobs must be done before the other jobs. In addition, each job also has a specific order of performing the tasks of that job. This order is referred to as a precedence constraint.

## 1.5 Organization of the thesis

Thesis is organized as follows:

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

Chapter 3 describes the scheduling problems as encountered in the literature. It presents the representation of schedule, Graham's law of scheduling problem, types of scheduling problems and some application areas of scheduling problems in operating system is also provided.

Chapter 4 describes the different solution strategies for scheduling problems.

Chapter 5 describes the single machine scheduling and its importance in scheduling and also describe the list of solution mentioned in past.

Chapter 6 describes the problem and some solution approach related to problem.

Chapter 7 describes the methodologies used in our dissertation and describes the comparison of two enumerative algorithm dynamic programming and branch and bound algorithm input is given by random number generator.

Chapter 8 describes the conclusion and further research.

# CHAPTER 2

# COMPUTATIONAL COMPLEXITY

## 2.1 Complexity Theory

Complexity theory is a part of *theory of computation* dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it takes to solve a problem) and *space* (how much memory does it take to solve a problem).Other resources can also be considered, such as how many parallel processors are needed to solve a problem in parallel. Complexity theory differs from *computability theory*, which deals with whether a problem can be solved at all, regardless of the resources required [6].
One of the major goals of *complexity theory* and algorithm analysis is to measure the performance of algorithms with respect to their computation time. The time complexity of an algorithm is the number of steps that it takes to solve an instance of the problem as a function of input size using the most efficient algorithm. The running time of an algorithm is said to be O(h(n)) if for a positive number c>0 there exists an implementation that terminates after at most c.h(n) for all n≥$n_0$.The time complexity of an algorithm is the smallest function such that the algorithm has running time O(h(n)).The time complexity T(k) of a problem $\prod$ is the minimal time complexity of all algorithms so that for some c>0 and $k_0 \epsilon z^+$ it holds T(k)≤c.h(k) for all k≥$k_0$.Remark that, the existence of this minimality in general is not guarantee and it is in fact one of the focal points of research in complexity theory. Obtaining the lower bounds for the complexity of a problem is harder; however upper bounds are usually obtained.
A polynomial algorithm is the one whose time complexity function T(k) ϵ O(h(n)),where h is some polynomial and n is the input length of an instance I.A computational problem is called polynomially solvable if there is a polynomial time algorithm solving it.

If time complexity function cannot be bounded by polynomial function, it is called exponential time algorithm.
The *complexity theory* provides a framework in which computation problems are studied so that they can be classified as “easy” or “hard”. Here we focus the main points of such theory.

A *polynomial time (polynomial)* algorithm is one whose time complexity function is O (p (k)), where p is some polynomial and k is the input length of an instance. Each algorithm whose time complexity function cannot be bounded in that way is called *exponential time algorithm*. Generally the problems with polynomial time algorithm are called *easy* problems with exponential time complexity are called *hard* problems [7].

## 2.2 Algorithms and Complexity

If computer problem solving can be summed up in one word, it is demanding! Problem solving is an intricate process requiring much thought, careful planning, logical precision, persistence and attention to detail. At the same time, it can be challenging, exciting and satisfying experience with considerable room for personal creativity and expression. If computer problem solving is approached in this sprit, then the chances of success are greatly amplified.

The computer solution to a problem is a set of explicit and unambiguous instructions expressed in a programming language. This set of instruction is called a program. Program may also be thought of as an algorithm expressed in programming language. An algorithm therefore corresponds to a solution to a problem.

### 2.2.1 Algorithm

An algorithm is a procedure for solving a problem (i.e. giving an answer).we will say that an algorithm solve the search problem, if it finds a solution for any instances I. In order to keep the representation of algorithms easily understandable, we follow a structural programming such as case statement, or loop of various kinds. Functions or procedures may also be called an algorithm. Parameter may be used to import data or to export data from the algorithm. Besides these, we also use mathematical notations such as set theoretic notations. In general, an algorithm consists of two parts: a head and a method. The head starts with the keywords algorithm followed by identifying number and optimality, a descriptor (a name or a description of the purpose of algorithm) references to the authors of the algorithm. Input and output parameters are omitted in case where they are clear from the context. In other case, they are specified as a parameter list. In even more complex case, two fields input (instances) and output

(Answer) are used to describe the main idea of the algorithm. The method part is block of the instructions.

### 2.2.2 Computational 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.

### 2.2.3 Time 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.

## 2.3 Decision /Recognition Problem

Optimization problems can rearranged in such a way that the solution is a “yes” or “no” answer (e.g. “is there a schedule for a given set of jobs that generates a number of tardy jobs less than a predetermined number say |L|?”).

If the recognition version of the problem is answered with "yes", the underlying instance is called a certificate (for the above example the certificate would be the schedule that achieves a number of tardy jobs less than |L|).

In other words**,** much of complexity theory deals with decision problems. A decision problem is a problem where answer is always YES/NO. For example a problem PATH related to shortest path problem is, "Given a path G= (V, E), two vertices u, v ∈ V and non-negative integer k, does a path in G between u and v whose length is at most k?". If I = (G, u, v, k) is an instance of this shortest path problem, then PATH (I) = "yes" if a shortest path from u to v has length at most k, and PATH (I) = "no" otherwise, [7].

## 2.4 Optimization Problems

We encounter many problems where there are many feasible solutions and our aim is to find the feasible solution with best value. This kind of problem is called optimization problem. For example given the graph G, and the vertices u and v find the shortest path from u to v with minimum number of edges. The NP completeness does not deal with optimization problems; however we can translate the optimization problem to the decision 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.

## 2.5 Abstract Problems and Encoding

Abstract problem A is the binary relation on set I of problem instances, and the set S of problem solutions. For example minimum spanning tree of a graph G can be viewed as a pair of the given graph G and MST graph T. Many abstract problems are not decision problems, but rather optimization problems in which some value must be minimized or maximized.

Encoding of a set S is a function e from S to the set of binary strings. With the help of encoding, we define the concrete problem as a problem with problem instances as the set of binary strings i.e. if we encode the abstract problem, then the resulting encoded problem is concrete problem.

So, encoding as a concrete problem assures that every encoded problem can be regarded as a language i.e. subset of $\{0, 1\}^{*}$.

## 2.6 Reducibility

A useful tool in studying the relationship between members of a class is the translation or mapping of one to another. If we can translate one set into another, we can often deduce properties of one by the properties that we know the other processes. A problem $A_1$ can be reduced into another problem $A_2$ if any instances of $A_1$ can be rephrased as an instances of $A_2$,the solution to which provides a solution to the instances of $A_1$ [8].For example, the problem of solving linear equation in an indeterminate x reduces to the problem of solving quadratic equations. Given an instance ax +b=0, we transform it to $0x^2$ +ax +b=0, whose solution provides to the solution to ax +b=0.Thus if problem $A_1$ reduces to another problem $A_2$, then $A_1$ is , in sense, no harder to solve than $A_2$. This notion can be represented as $A_1 \leq pA_2$. The fig 3.1 below shows this strategy.

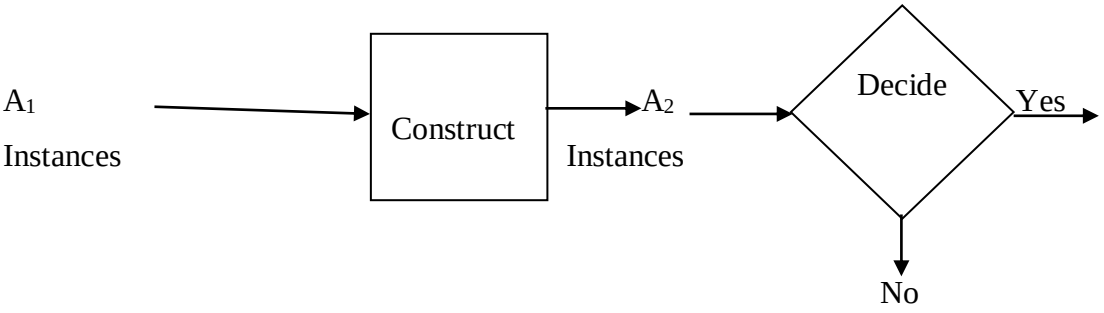



*Figure 2.1: Problem reduction[8]*

## 2.7 Complexity Classes

### 2.7.1The Class P

The class P consist of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in size of input i.e. the time complexity function is polynomial [7].In other word, a decision problem is in the class P if there exists an algorithm that solves any instances of size n in O ($n^k$) time, for some integer k. So P is just the set of tractable decision problem: the decision problem for which we have polynomial time algorithms.

Example 2.1 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.

### 2.7.2The Class NP

The complexity class NP is the set of decision problem that can be solved by non-deterministic polynomial time. Equivalently we can say that class NP consists of all those decision problems whose positive solution can be verified in polynomial time. The important of this class of decision problems is that it contains many interesting searching and optimization problems where we want to know if there exists a certain solution for a certain problem or whether there exists a better solution.

Example 2.2 The Travelling salesman problem where we want to know if there is a shorter route that goes through all the nodes in a certain formula in propositional logic with propositional variables is satisfiable or not.

### 2.7.3 P $\subseteq$ NP

A P problem is always also NP. If a problem is known to be NP and a solution the problem is some how known, then demonstrating the correctness of the solution can always be reduced to a single P verification. If P and NP are equivalent, then the solution of NP problems requires an exhaustive search.

### 2.7.4 Big Question: Is P=NP?

The question of whether P is the same set as NP is the most open question in theoretical computer science and modern mathematics. The question of the equality of these two classes was originally posed in a letter from Kurt Gödel to J. von Neumann. There is even a $ 1,000,000 prize for solving it [6].

If any problem A $\epsilon$ P, then A $\epsilon$ NP, since there is a polynomial time algorithm to decide A, the algorithm can be easily converted to a two argument verification algorithm that simply ignores any certificate and accepts exactly those input it determines to be in A. Thus, we can say **P** $\subseteq$ **NP**. The definition of **NP-completeness** leads to the sense that the **NP-complete** problems are ones most likely not to be in **P.** The reason is that if we could find a way to solve an **NP-complete** problem quickly, then we could use that algorithm to solve all **NP** problems quickly. Most

theoretical computer scientist believes that P≠NP, which leads to the relationships among P, NP, NP-complete exist in following figure. But if someone finds a polynomial time algorithm for NP-complete problem, it will prove P=NP. Nevertheless, no polynomial time algorithm for any NP complete has yet been discovered.

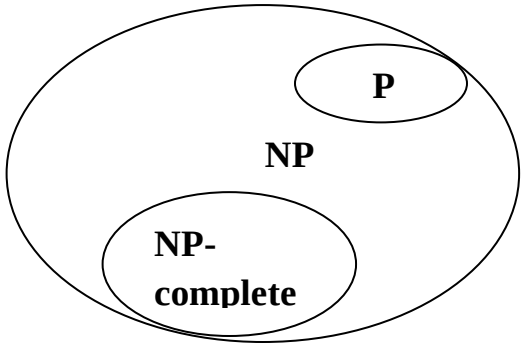



*Figure 2.2: Relationship between P, NP and NP-complete*

If *P=NP,P* would encompass the *NP* and *NP-complete* areas.

### *2.7.5 NP-complete* problems

*NP-complete* problems are the hardest problems in *NP*. It is also defined as a decision problem D is *NP-complete* if it is in *NP* and if every other problem in *NP* is reducible to it [8].”Reducible” here means that for every *NP* problem L, there is a polynomial time algorithm which transforms instances of L into instances of D, such that two instances have the same truth values. As a consequence, if we had a polynomial time algorithm for D, we could solve all *NP* problems in polynomial time.

*Example 2.3 Boolean satisfiability problem*

The Boolean satisfiability problem (SAT) is a decision problem considered in complexity theory. An instance of the problem is defined by Boolean expression written using only AND, OR, NOT, variables and parentheses. The question is: given the expression, is there some assignment of TRUE and FALSE values to variable that will make the entire expression true? The SAT is NP-complete. In fact it was first known *NP-complete* problem [8].

### *2.7.6 NP-hard* problem

A problem is *NP-hard* (non-deterministic polynomial time hard) if solving it in polynomial time would make it possible to solve all problems in class *NP* in polynomial time. That is a problem is *NP-hard* if an algorithm for solving it can be translated into one for solving any other NP problem.NP hard therefore means "at least as any *NP* problem".

### *2.7.7 Co-NP* class

Co-NP is the set containing the complement problems (i.e. problems with the YES/NO answers reserved) of *NP* problems. It is believed that the two classes are not equal; however it has not yet been proven. It has been shown that if these two complexity classes are not equal, then it follows that no *NP-complete* problems can be in *Co-NP* and no *Co-NP complete* problems can be in *NP*.

### *2.7.8 Co-NP-complete* problems

In complexity theory, the complexity class *Co-NP-Complete* is the set of problems that are hardest problems in *Co-NP*, in the sense that they are the ones most likely not to be in P. If we can find a way to solve a *Co-NP-Complete* problem quickly, then we can use that algorithm to solve all *Co-NP* problems quickly.

A more formal definition: A decision problem A is *Co-NP*-Complete if it is in *Co-NP* and if every problem in *Co-NP* is many-one reducible to it. This means that for every *Co-NP* problems L, there exists a polynomial time algorithm which can transform any instances of L into an instance of A with the same truth values. As a consequence, if we had a polynomial time algorithm for A, we could solve all *Co-NP* problems in polynomial time.

### 2.7.9 Famous Complexity Classes

The following are the some of the classes of problems considered in complexity theory, along with informal definitions.

| | |
|---|---|
| *P* | Solvable in polynomial time |
| *NP* | YES answers checkable in polynomial time |

| | |
|---|---|
| *Co-NP* | No answers checkable in polynomial time |
| *NP-Complete* | The hardest problems in *NP* |
| *Co-NP-Complete* | The hardest problem in *Co-NP* |
| *NP-hard* | Either *NP-Complete* or harder |

*Table 2.1 Famous Complexity Classes*

# CHAPTER 3

# SCHEDULING THEORY

In more general we say that scheduling is an allocation of one or more time intervals to each job on one or more machines. A scheduling is called optimal if it minimized a given objective function mean to establish an assignment of resources to consumers for a certain period of time in a way that a certain objective is optimized. And the policy used to determine this assignment is called scheduling algorithm.

Scheduling theory is excessively used in computer manufacturing to schedule the jobs in CPU, memory, printing buffer, spooling and other devices for processing jobs. The multiprogramming characteristics of computer due to the good scheduling jobs in the CPU because of the CPU can only process one job at a time. In this case the objective function is to maximize the CPU utilization [9].

## 3.1 Representation of Scheduling Problem

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. Which may be machine oriented or job oriented. 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.

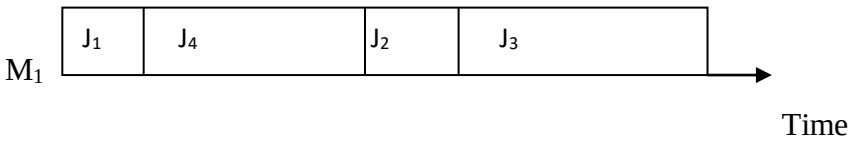



*Figure 3.1: Gantt chart for schedule of four jobs in single machine*

## 3.2 Classes of Schedules

Certain classes of schedules are introduced [10].

- A schedule is called non-delay if no machine is kept idle when there exists a job available for processing.
- A schedule is called active, if no operation can be completed earlier by changing the job orders without delaying any other operation.
- A schedule is called semi-active, if no operation can be completed earlier without changing the sequence.

Therefore the following properties hold:

***Non-delay schedule*** $\Rightarrow$ ***active schedule*** $\Rightarrow$ ***Semi-active schedule***

$(\not\Leftarrow)$ $(\not\Leftarrow)$

The following figure illustrates the connection between the above introduced classes of schedules.

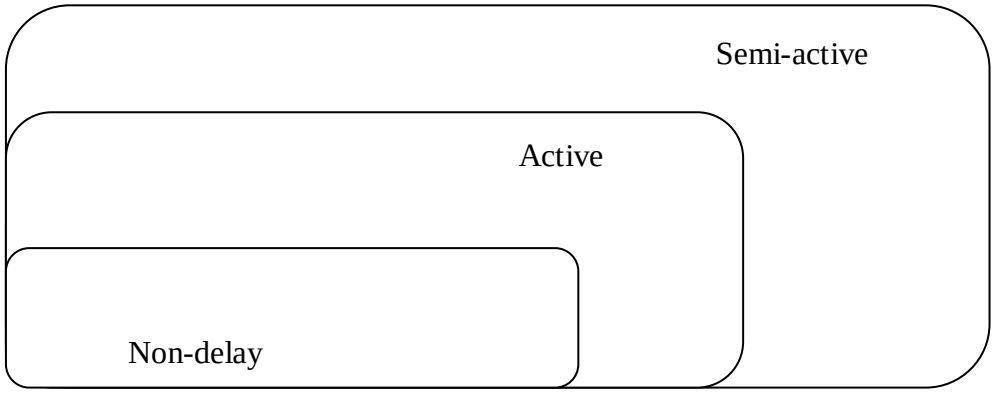



*Figure 3.2 Classes of schedule[10]*

## 3.3 Types of Scheduling Problem

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

### 3.3.1 Single Machine

Single machine models are important for various reasons. The single machine environment is very simple and a special case of all other environments. Single machine models often have properties that neither machine in parallel nor machine in series has. The results can be obtained for single machine models not only provide insights into the single machine environments, they also provide a basis for heuristics that are applicable to more complicated machine environments are often decomposed into sub problems that deal with single machines. For example a complicated machine environment with a single bottleneck may give rise to a single machine model [2].

### 3.3.2 Parallel Machine

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.

### 3.3.3 General Shop Scheduling Problem

In this section we will discuss general shop scheduling problems like open shop problem, job shop problem, mixed shop problem and super shop problems. Which are widely used for modeling production processes? All of these problems are special cases of general shop problems [5].

### 3.3.4 Flow Shop Problems

The most well known shop scheduling problem is the flow shop. Here it is assumed that each job $J_j$ consists of M operations $O_{1j}$, $O_{2j}$, ……., $O_{mj}$ with processing times $P_{ij}$ to be performed in this order, operation $O_{ij}$ being processed on machine $M_j$. In other words, each job $J_j$ is first processed on machine $M_1$, then on machine $M_2$,and so on, until it is processed on machine $M_m$. In what follows, the order in which a job has to pass the machine is called the processing route. Thus, in the flow shop all jobs are given the same processing route ($M_1$, $M_2$,……,$M_m$). The problem is to find a job order for each machine.

### 3.3.5 Job Shop Problem

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.

### 3.3.6 Open Shop Problem

An open shop problem is a special case of the general shop in which each job i consists of m operations $O_{ij}$ (j=1, 2,……,m) where $O_{ij}$ must be processed on machine $M_j$ and there are no precedence relation between the operation.

A schedule is said to be non-preemptive if each operation is executed continuously from start to completion. A schedule is preemptive if the execution of any operation may arbitrarily often be interrupted and resumed at a time, the periods in which the operation of a given jobs are performed may be interleaved in time.

### 3.3.7 Mixed Shop Problem

These three basic models can be generalized by combining some or all of them. For example, combining the flow shop and open shop, we obtain the model which is known as mixed shop. More precisely, for the mixed shop, it is assumed that the set J of jobs is partitioned into two non-empty subsets $J_0$ and $J_1$. The jobs of the set $J_0$ have non fixed orders of their operations (as in an open shop), while all jobs of the set $J_1$ have the processing route ($M_1$,$M_2$,……,$M_m$) (as in a flow shop).

### 3.3.8 Super Shop Problem

One of the most general shop scheduling models, which covers all the previous ones, is called the super shop. This is obtained as a result of combining the open shop and the job shop. According to that model, set J is partitioned into r+1 subsets $J_0$, $J_1$, $J_2$, ……., $J_r$. The jobs of set of $J_0$ have non fixed orders of processing their operations (as in an open shop), while the jobs of a set $J_q$, $1 \leq q \leq r$, have the processing route $L_q$; some machines of set M may not occur in a

sequence $L_q$, while, on the other hand, some of them may occur more than once (as in a job shop).

### 3.3.9 Static and Dynamic

If all the data of the problem are known at the same time we speak of a static problem. For some problems a schedule may have been calculated and being processed when new operations arrive in the system. Then the foregoing schedule has to re-establish in real-time. These problems are said to be dynamic. In other words static and dynamic are defined as follows: Depending on the release times, a model (problem) is referred to as a static model if the release times of all jobs are zero ($r_j$=0 for every j). Problems with varying nonzero release times are called dynamic.

### 3.3.10 Stochastic and Deterministic

A models that assume some of data to be randomly fluctuating (e.g. defects, break-downs, set-ups, etc as stochastic date [16].

A model is deterministic if all data defining a problem is available and known with certainty.

## 3.4 Classification (The Three Field α|β|γ Notation)

There are varieties of classes of scheduling problems, which differ in their complexity. Also the algorithms developed are quite different for different class of scheduling problems. Classes of scheduling problems are specified in terms of a three field classification α|β|γ, where

- α specifies the machine environment.
- β describing the job and resource characteristics.
- γ denoting the optimality criteria.

This classification scheme was introduced by Graham et al in 1979 [5]

### 3.4.1 Machine Environment (α):

The machine environment is characterized by a string $\alpha=\alpha_1.\alpha_2$ where,

- $\alpha_1$ describing the type of machine used or their arrangement , respectively
- $\alpha_2$ describing the number of machines or the number of processing steps, respectively.

### 3.4.1.1 Machine Types and Arrangements

The possible values of $\alpha_1$ are Φ,P,Q,R,PMPM,QMPM,G,X,O,J,F. If $\alpha_1 \epsilon$ (Φ,P,Q,R,PMPM,QMPM),where Φ denotes the empty symbols(thus $\alpha=\alpha_2$ if $\alpha_1=$ Φ),then each job $J_j$ consists of a single operation.

If $\alpha_1=$ Φ, then each job must be processed on a specified (dedicated) machine, single machine.

If $\alpha_1 \epsilon$ (Φ,P, Q, R),then we have parallel machines.

If $\alpha_1$=P, then we have identical parallel machine.

If $\alpha_1$=Q, then there are uniform parallel machine.

If $\alpha_1$=R, then there are unrelated parallel machine.

If $\alpha_1$=PMPM, then we have multi-purpose machine and identical speed.

If $\alpha_1$=QMQM, then we have multi-purpose machine with uniform speed.

If $\alpha_1 \epsilon$ (G, X, O, F, J) we have the multi-operational model.

If $\alpha_1$=G, then it represent a general shop model.

If $\alpha_1$=J, then it represent the job shop model, there are m dedicated machine available. Each job has its identical flow pattern.

If $\alpha_1$=O, then it represent the open shop model. The machine orders and the job orders can be chosen arbitrarily.

If $\alpha_1$=F, then it represent the flow shop model. All jobs have identical flow patterns and each job has to seize each machine exactly once.

If $\alpha_1$=X, then it represent mixed shop model.

In my dissertation we consider only single machine model.

### 3.4.1.2 Machine Number

Parameter $\alpha_2 \epsilon \{\Phi, M\}$ describes either the number of machines, in case of parallel processors and one operation, or the number of processing steps , in case of dedicated processors with more than one operation.

If the number of machines is arbitrary we set $\alpha_2 = \Phi$.

### 3.4.2 Job Characteristics (β)

The job characteristics are specified by a set β containing at most six elements $\beta_1$, $\beta 2$, $\beta 3$, $\beta 4$, $\beta 5$, $\beta 6$.

- $\beta_1$ indicate**s** whether preemption is allowed. If preemption is allowed the processing of a job on a machine can be interrupted and continued later on, and we set $\beta_1 = pmtn$**,** otherwise $\beta_1$ does not appear in β.
- $\beta_2$ describes precedence relation between jobs which may be represented by an acyclic directed graph and we set $\beta_2 = prec$. Sometimes we will consider the scheduling problems with restricted precedence given by chains, an intree, an outtree, a tree or a series parallel directed graph. In these cases we set $\beta_2$ equals to chains, intree, outtree, and sp-graph.
- If $\beta_3 = r_j$ then release dates may be specified for each job. If $r_j = 0$ for all jobs then $\beta_3$ does not appear in β.
- $\beta_4$ specifies restriction on the processing times or on the number of operations. If $\beta_4$ equals to $P_{ij} = 1$ then each job (operation) has unit processing requirements.
- If $\beta_5 = d_j$ then a deadline $d_j$ is specified for each job $J_j$ ,i.e job $J_j$ must finish not later than time $d_i$
- $\beta_6$=bath indicates a batching problem. A batch is a set of jobs, which must be processed successively on a machine and batching problem is to group the jobs into batches and to schedule these batches.

### 3.4.3 Optimality Criteria (γ)

The scheduling problem is to find a feasible schedule which minimizes the total cost function. We denote the finish time of job $J_j$ by $C_j$ and associated cost by $f_j(C_j)$.If the function $f_j$ are not specified ,we set $\gamma = f_{max}$ or $\gamma = \sum f_j$. The most common are the makespan

max{$C_j$|j=1,……..,n}, total flow time$\sum_{j=1}^{n} C_j$ and weighted (total) flow time $\sum_{j=1}^{n} W_j C_j$.In these case we write $\gamma=C_{max}$,$\gamma=\sum C_j$,and $\gamma=\sum W_j C_j$ respectively. Following are the main objectives function in scheduling for each job $J_j$,let release time be $r_j$, due date be $d_j$ and weight be $w_j$.

| | |
|---|---|
| Completion time | $C_j$ |
| Flow time | $F_j=C_j-r_j$ |
| Lateness | $L_j=C_j-d_j$ |
| Tardiness | $T_j=\max\{0,C_j-d_j\}$ |
| Earliness | $E_j=\max\{0,d_j-C_j\}$ |
| Unit penalty | $U_j = \begin{cases} 0, & if\ C_j \le d_j, \\ 1, & Otherwise \end{cases}$ |

Other objective functions are

| | |
|---|---|
| Total completion time | $\sum C_j$ |
| Total tardiness | $\sum T_j$ |
| Total weighted completion time | $\sum W_j C_j$ |
| Total weighted tardiness | $\sum W_j T_j$ |
| Number of tardy jobs | $\sum U_j$ |
| Weighted number of tardy jobs | $\sum W_j U_j$ |

In my dissertation we consider the $\sum W_j U_j$ objective function and my aim is to minimize the weighted number of tardy jobs.

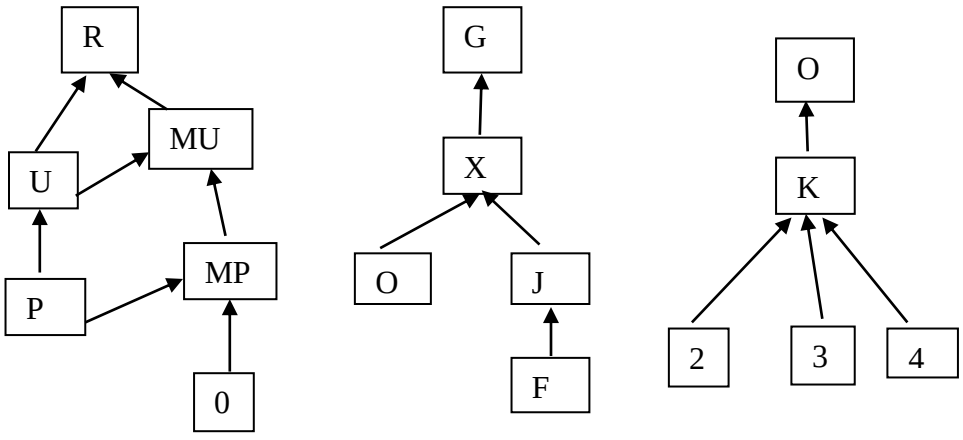
In γ field, the formula or a short symbol for denoting the objective function is simply written. For example we can write $\sum W_j U_j$ to indicate the weighted number of tardy jobs has to be minimized. Here are some example of three field notation $1|r_j|\sum U_j$ denote the single machine schedule where release date is given and the objective to minimize the number of tardy jobs.

But in my dissertation we consider the problem in three field notation as $1||\sum W_j U_j$, a single machine problem with no release time and objective function is to minimize the weighted number of tardy jobs.

Another example is that $J3|P_{ij}=1|C_{max}$is the problem of minimizing maximum completion time in three machine job shop with unit processing times.

## 3.5 Simple Reduction between Scheduling Problems

If in description of scheduling problem we replace F by J, we get a simple reduction because the flow shop is a special case of job shop. Similarly we get a simple reduction, if we replace tree by prec. These simple reduction are shown by the reduction graph $G_i$ (i=1,…..,7) in fig 3.3



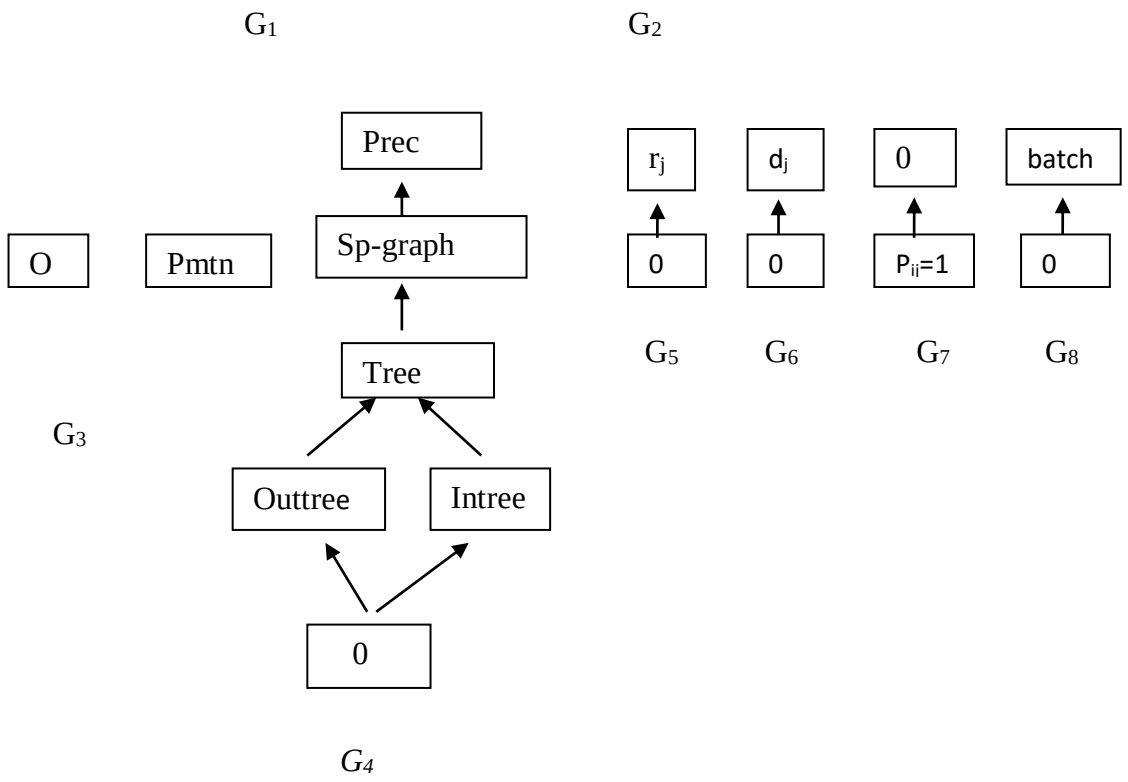



*Fig 3.3 Reduction between scheduling problem [5]*

There are similar reductions between objective functions. These relation are shown in figure 3.3$\sum f_j$ reduces to $\sum W_j f_j$ by setting $W_j=1$ for all j,$L_{max}$,$\sum C_j$ and $\sum W_j C_j$ reduces to $L_{max}$, $\sum T_j$ and $\sum W_j T_j$ respectively by setting $d_j=0$,for all j.

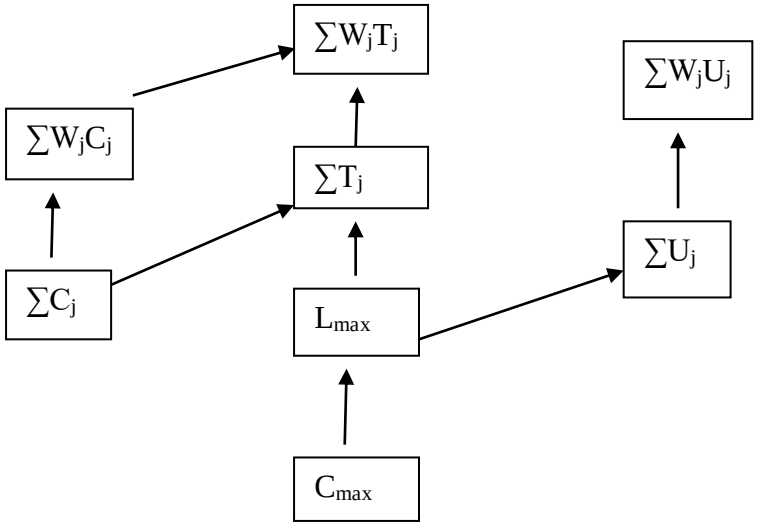



*Figure 3.4 Relation between objective functions [5]*

## 3.6 Some Application Areas of Scheduling Problem

Scheduling plays an important role in most manufacturing and production system as well as in most information processing environments. Scheduling problems are encountered at all levels and in all sectors of activity. Scheduling can be difficult from a technical as well as from implementation point of view. Generally we can distinguish between those of manufacturing production and those in computer in computer systems or project management

### 3.6.1 Problems Related To Production

We encounter scheduling problems in Flexible Manufacturing System (FMS).Numerous definitions of an FMS are found in the literature. Lu and MacCarthy [12], states: "An FMS comprises three principal elements: computer controlled machine tools, an automated transport system and a computer control system". Besides, this very broad problem encompasses other problem related to Robotic Cell Scheduling and Scheduling of Automated Guided Vehicles (AGV). Electroplating and chemical shops have their own peculiarities in scheduling problems. The shops are characterized by the presence of one or more traveling cranes sharing the same physical area and which are ordered to transport the products for treatment in tanks. In general, the soaking time in a tank is bounded by a minimum and maximum , transport time is not negligible and the operation must be carried out without waiting time. These problems are very common in industry and the "simple" cases (mono-robot, single batch tanks, etc) have been solved by now.

Scheduling problems in car production line, so called Car Sequencing Problems, are encountered in assembly shops where certain equipment must be assembled in the different models of vehicles. These problems have constraints and peculiarities of their own. Knowing a sequence of vehicles undergoing treatment, the problem is to determine the type of next vehicle programmed. We have to take account of a group of constraints connected principally to assembly options for these vehicles and to the limited movement of the tools along the production line.

### 3.6.2 Scheduling Problem in Operating System

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 [13].

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.

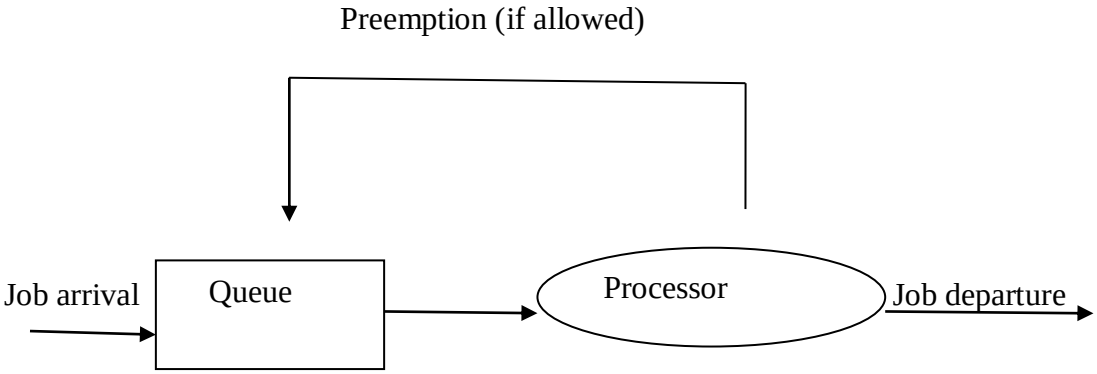



*Figure 3.5 The basic queuing model [13]*

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.

### 3.6.3 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* [14].

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.

## 3.7 Just-In-Time and Real-Time System

Just-In-Time (JIT) is the name used to describe as manufacturing system where the parts which are needed to compute the finished products are produced or arrive at the assembly site as they are needed. Just-In-Time is a Japanese manufacturing management method developed in 1970s.It was first adopted in Toyota manufacturing plants by *Taiichi Ohno*. The main concern at that time was to meet consumer demands.

The main concept of penalizing jobs both for being tardy and for being early has proven one of the most important and fertile research topic in operations research. Sequencing different products with even distribution under Just-In-Time production for minimization of earliness and tardiness penalties is a challenging non-linear integer programming problem. The purpose of Just-In-Time is to reduce cost by eliminating waste. In sales the Just-In-Time concept is realized by producing only solvable products or part in salable quantities. The main goal of Just-In-Time approach is to sequence small batches of variety of parts types in order to satisfy customers demand for them without holding excessive inventories or incurring large shortage. A sequence with this goal is termed as a balanced schedule, Miltenburg and Sinnamon [16]. In some special cases, this sequence will be optimal for level schedule problem for mixed model assembly line, Monden [16]. The level schedule problem is concerned with keeping as constant as possible the rate of usage of component parts going into part type being assembled.

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.[17], [11].

Real-time computing system plays a vital role in our society-controlling laboratory experiments, automobile engines, nuclear power plants, flight systems, and manufacturing processes and the spectrum of their complexity varies widely from the very simple to very complex. 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* [13] provides an introduction for real-time scheduling problems in operating systems.

# CHAPTER 4

# ANALYZING AND SOLVING SCHEDULING PROBLEMS

Since numerous techniques for analyzing and solving scheduling problems under different constraints are known, a complete overview and description is cumbersome. The main aim of this section is to introduce a general scheme of how much to approach scheduling problems, to roughly classify the most common solving techniques and to give some examples of certain frequently used methods.

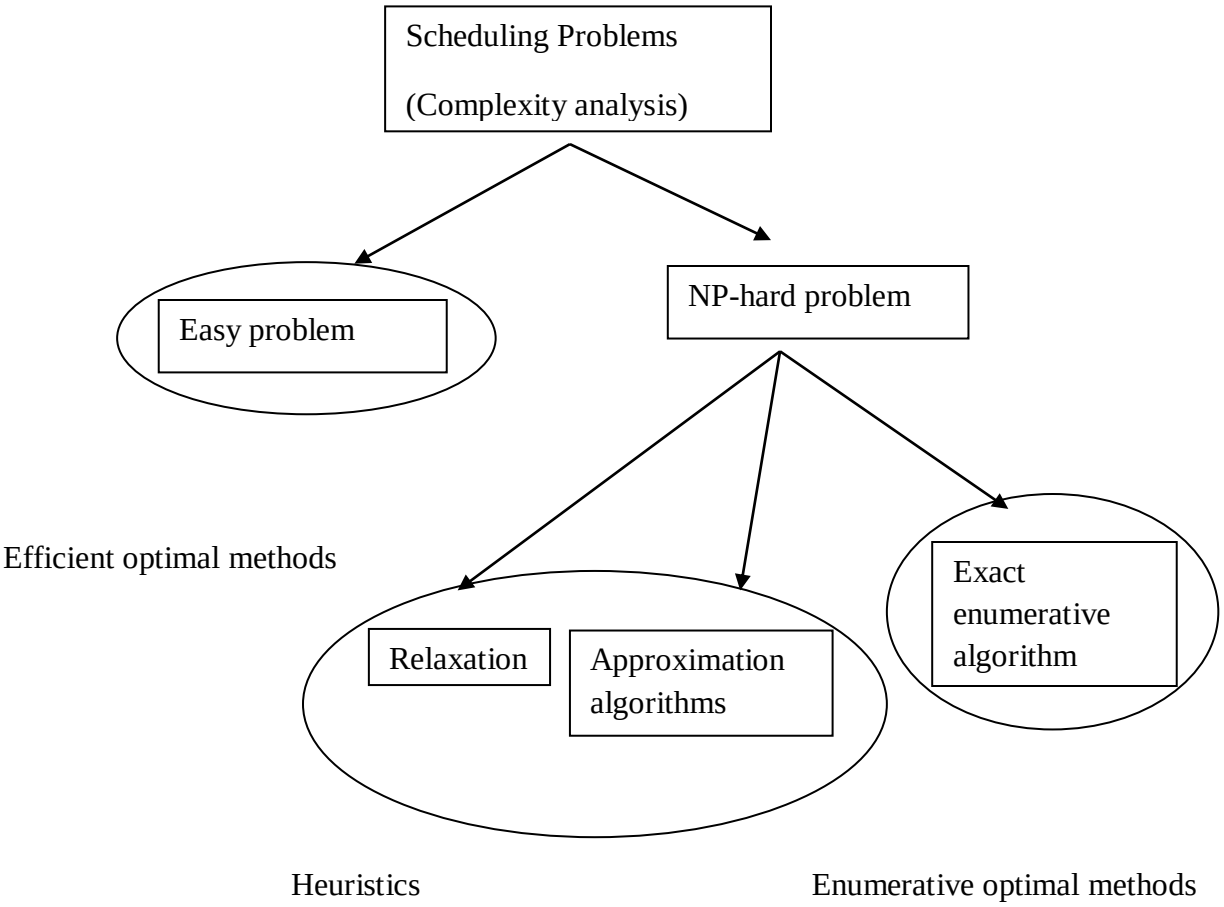



*Figure 4.1 Analyzing scheduling problems [12]*

Figure 4.1 shows a schematic approach of how to analyze scheduling problems proposed by *Blazewicz et al.[14]*.The three typical groups which – according to *MacCarthy and Liu [12]*- comprise most of the solving technique are outlined.

Referring to the above mentioned groups of solving methods, it is distinguished between:

## 4.1 Efficient optimal methods

This group includes the methods/algorithms typically used for the class of polynomially solvable problems. Thus, they guarantee an optimal solution within polynomial time even for large problems. Most of the optimal algorithms are dedicated only to a specific kind of problem or to small class. Hence, they are rare and applicable only for a few and quite fundamental issues. Even if such methods exist, it might sometimes be more useful to rely upon other approaches, like good heuristics, if the exact optimal algorithm is of too high complexity.

## 4.2 Enumerative Optimal Methods

Enumerative methods typically use a partial enumeration of the set of all solutions that are possible macCarthy and Liu. Methods of implicit enumeration variety "consider certain solutions only indirectly, without evaluating them explicitly Blazewicz et al. [14].The main approaches are:

- Dynamic programming
- Branch and bound method

Both are exponential in nature but promising mainly for problems of smaller size. Especially dynamic programming algorithms can often be constructed to obtain pseudopolynomially bounded approaches.

### 4.2.1 Dynamic Programming

Dynamic programming is a powerful algorithm paradigm. Problems are solved by identifying smaller sub problems and by solving one sub problem after the other, starting with smallest and using the answers to the small problems until the initial problem is solved [18]. In other words, dynamic programming, like the divide and conquer method, solves problem by combining the

solutions to sub problems. Divide and conquer algorithms partition the problem into independent sub problems, solve the sub problems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the sub problems are not independent, that is, when sub problems share subsubproblems. In this context, a divide and conquer algorithm does more work than necessary, repeatedly solving the common subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub problem is encountered. Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (maximum or minimum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

An example of Baker is used to outline the characteristics of dynamic programming procedures [19]

- A set J of j=1, ….., n jobs is to scheduled on a single machine. S denotes a subset of J and $\overline{S}$ its complement, i.e. the set of jobs not contained in S. Schedules are to be constructed such that all jobs in $\overline{S}$ precede every job in S.
- $C_j$ is the completion time of job j and C ($\overline{S}$) the total time required to process all jobs in $\overline{S}$.
- Let z be the performance measure with an additive structure like $\sum_{j=1}^{n} g_j(C_j)$, with $g$j(Cj) being the marginal contribution of job j to the overall cost in dependency of its completion time $C_j$. If z is the number of tardy jobs, $g$j(Cj) would be 1 if job j is scheduled tardy and 0 otherwise.
- The aim is to minimize the value of z.

Most applications of dynamic programming have following characteristics [20].

1. The initial problem can be divided into several stages. A decision is required at each stage. Considering the example, a stage k could be characterized by the size of the subset S, i.e. the number of jobs to be schedule at this stage. Hence, in the first stage the task consists of schedule one job, in the second of scheduling two jobs and so on.

2. Each stage has a number of associated states, the information needed at any stage to make an optimal decision. In the mentioned example the states are the possible subsets S. If J consists of four jobs j=1, ……. ,4 and one looks at the second stage, i.e. a pair of two jobs is to be considered, the states are:{1,2},{1,3},{1,4},{2,3},{2,4},{3,4}
3. Decisions have to be made in any stage. Decisions describe how the state at the current state is transformed into the state at the next stage. In our example a decision is simply the subsequent job to be chosen in the next stage's S, determining the next stages.
4. Principle of optimality: Given the current state, the optimal decision for each of the remaining stages must not depend on previously reached states or previously made decision.

   Since the overall problem consists of several stages, there must obviously be recursion that relates the costs or rewards from one stage to the other.

Dynamic programming is used to solve scheduling problems such as, $1||\sum w_i U_i$, $1|batch|\sum C_i$ and so on.

### *4.2.2* Branch and bound (B&B) Algorithm

Branch and bound algorithm is another method for solving combinatorial optimization problems. It is based on the idea of intelligently enumerating all feasible solutions. We assume that the discrete optimization problem P to be solved is a minimization problem. P may be identified with the corresponding set S of feasible solutions. Generally, B & B methods decompose a complex problem into multiple subproblems and utilize known methods to solve the easier subproblems [21]. Another advantages of the B & B approaches is that although the worst case complexity is exponential, especially in situations where the search for the solution takes a lot of time the B & B procedure can be aborted at any stage using the best solution known so far [Comp. soric (2002,p.13). The name branch and bound denotes that this method is based on two main steps. The problem decomposition is the result of branching procedure leading to subproblems which are:

1. Mutually exclusive and exhaustive subproblems of the original.
2. Partially solved problems of the original.
3. Smaller problems than the original [19]

Since decomposing can be continued i.e. a problem is branched and its subproblems are further decomposed, a tree like form is obtained. Figure 4.2 shows the above things.

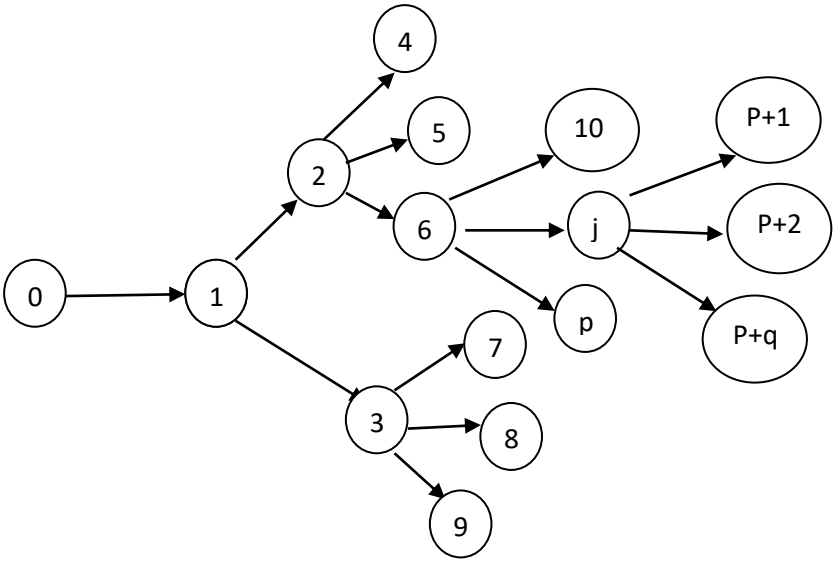



Figure 4.2 Branching tree-examples [19]

The main procedure is bounding which calculates lower and upper bounds and fathoms a branch if necessary. If one considers a minimizing the objective functions in an integer linear programming problem, for each subproblem a lower bound (noted as LB) might be calculated by ignoring integer constraints, allowing those variables to take real values. This kind of relaxation is known as linear programming relaxation (LP-relaxation). Additionally, an initial upper bound (noted as UB) is computed by a fast heuristic or arbitrarily determined to be infinite. During the algorithm, the current best LB obtained by a solution which feasible to subproblem as well as to the underlying main problem becomes new UB [20]. At each node within the search tree it is checked if the current branch should be partitioned further. If a branch is no more considered in the ongoing algorithm, it is said to be fathomed. Obviously, due to not partitioning the fathomed branches any further, the enumeration process can be curtailed [19]. Following reasons cause a branch to be fathomed [19].

- No feasible solution is available for the relaxation.
- The LB received by relaxation is larger or equal than the current UB.

- The relaxation's solution is feasible for the subproblem and the underlying main problem, and its objectives value is even better than the current UB. In the latter case, the LB becomes the new UB.

## 4.3 Heuristic Methods

Heuristic are kind of rule-of-thumbs techniques which approximate the optimal solution but cannot guarantee its finding in all cases [20]. They are applied for both, solving specific problems to optimality and providing a fast and simple but acceptably good solution for hard problems [22]. Heuristics are typically judged by their "goodness" of approximation; their performance, often in a west case scenario, compared with the optimal solution (if known) as ratio or difference. Heuristics with analytically evaluated accuracy are referred to as approximation algorithms [14]. In this context, a ρ-approximation algorithm generates a result of at most ρ times the optimal value in polynomial time [23]. Some heuristic algorithms and strategies for scheduling problems are

- Relaxation based heuristics
- Scheduling rules / priority rules
- Simulated annealing
- Tabu search
- Genetic algorithms

### 4.3.1 Relaxation:-

Relaxation restricts the universality of the problem considering only special types of input instances. Actually, it is not technique for solving given problem, rather a compromise made due to difficulties forwarded by the problem. In scheduling following types of relaxation are often used.

1. Allowing preempting :- Complexities of most of the scheduling problems can be reduced by allowing preemption. For example, the problem $1|r_j|\sum C_j$ is NP-Hard [24] but $1|r_j,pmtn|\sum C_j$ can be solved in O(nlogn) time, n is the number of jobs.
2. Allowing unit processing time:- E.g. the problem $1||\sum W_jU_j$ is NP-Hard [24], but the problem $1|p_j|\sum W_jU_j$ can be solved in O(nlogn) time [25].

3. Assuming equal release dates:- E.g. the problem $1|r_j|\sum L_{max}$ is NP-Hard but if $r_j=r$ for all jobs $J_j$ then it can be solved in $O(n^2)$ time [25].
4. Assuming certain precedence relation:- E.g. the problem $1|prec|\sum W_jC_j$ can be solved in O(nlogn) time [25].

### 4.3.2 Scheduling rules/priority rules

Scheduling rules can be defined as a rule that dictates which jobs among those waiting for service are to be scheduled in performance to the others [26]. Note that such a rule only indicates which job to be serving first, in contrast to job sequencing which orders all jobs in a queue due to specific attributes [27]. Scheduling rule is the most general prescription. It can comply with a single priority rule but can also be more complex combining several priority rules or one or more heuristics.

Priority rule is a simple function that assigns a priority as a number a value to each waiting jobs following a predetermined method. [26].The job with the highest priority the one with the lowest priority value is selected first. The FIFO rule prioritizing always the job that has arrived first is a typical example. [26].This rule is also known as dispatching rule. The jobs are arranged in a list according to some rule. The next job on the list is assigned to the first available machine. The following are some of the common rules.

#### 4.3.2.1 Random List

This list is made according to a random permutation.

#### 4.3.2.2 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.

#### 4.3.2.3 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.

#### 4.3.2.4 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.

#### 4.3.2.5 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.

#### 4.3.2.6 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.

#### 4.3.2.7 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

Dispatching rules are used in many contexts. For some easy problems, especially those in a single server environment, they enable an optimal solving, like for instance:

- The EDD-rule for $[1||T_{max}]$ and $[1||L_{max}]$, namely minimizing the maximum tardiness and lateness, respectively.
- The preemptive EDD-rule for minimizing the maximum tardiness and lateness with release time constraints and preemption ($[1|pmtn; r_j|T_{max}]$ and $[1|pmtn; r_j|L_{max}]$).
- The SPT-rule for minimizing the mean lateness ($[1||\bar{L}]$), ($[1||\bar{F}]$) and total completion time ($[1||\sum C_j]$),
- The WSPT-rule for minimizing the total weighted completion time ($1||\sum w_j C_j]$),
- The EDD-rule for minimizing the make-span for a single machine problem with release time constraints ($[1|r_j|C_{max}]$).

### 4.3.3 Simulated Annealing

Simulated annealing has its origin on the analogy between the annealing process of solids and the problem of solving combinatorial optimization problems. At the beginning, almost all solutions are accepted. Then, generally the “temperature” is dropped meaning that the mechanism of accepting new solutions is increasingly more selective. At the end, only the solutions that improve the objective function value are accepted. Each point in the search space has an energy associated with it, which indicates how good it is. The goal is to find the point

with minimum energy. The algorithm starts off at an arbitrary point; at each step chooses some neighbor of the current point and moves to that point with a certain probability. Neighbors are points that are close to each other in a function of the energy difference between two points and a global time-dependent parameter called temperature. Let $\Delta E$ be the difference in energy and $T$ be the temperature. If $\Delta E$ is negative then the algorithm moves to new point with probability 1. If not it does so with probability $e^{-\Delta E/T}$. This rule is deliberately similar to the Maxwell-Boltzmann distribution governing the distribution of molecular energies.

It is clear that the behavior of the algorithm is crucially dependent on the temperature: if $T$ is 0, it reduces to the greedy algorithm, always moving to a point of lower energy. If $T$ is infinity, it moves around randomly. At first $T$ is set to infinity, and it gradually decrease to zero ("cooling"). This enables the algorithm to initially get to the general region of the search space containing good solutions, and later hone in optimum. The exact annealing schedule, however, cannot be generally prescribed; it must be chosen depending on the problem.

It can be shown that, for any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimum solution approaches 1 as the cooling rate is decreased. This fact is, however, not particularly useful in practice, as at some point the time required to execute the algorithm will exceed the time required for a complete search of solution space. Simulated annealing can be very effective at finding good sub-optimal solutions.

**Algorithm 4.2**: [42] Simulated Annealing

1. *select an initial solution* $s$.
2. *for* $t=1$ *to* $\infty$ *do*

    *2.1. if* $T = 0$ *then return* $s$.

    *2.2.* $s'$ = *a randomly selected solution from* $N(s)$.

    *2.3.* $\Delta E = cost(s') - cost(s)$.

        *2.3.1. if* $\Delta E < 0$ *then* $s = s'$.

        *2.3.2. else* $s = s'$ *only with probability* $e^{-\Delta E/T}$.

### 4.3.4 Tabu Search

The basic Concept of Tabu Search is described in *Glover* [29]. 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.

### 4.3.5 Genetic Algorithm

Genetic Algorithms (GAs) were originally proposed by John H and Holland [30]. They are search algorithms that explore a solution space and mimic the biological evolution process. There are many GA implementations successfully applied to a great variety of problems. The main components of a genetic algorithm are as follows.

1 **Solution encoding:** A chromosomal representation of solutions.

2 **Initial populations:** Creation of an initial population of chromosomes.

3 **Fitness:** Measurement of chromosome fitness based on the objective function.

4 **Selection:** Natural selection of some chromosomes (parents) in the population for generating new members (children) in the population.

5 **Genetic operators:** Genetic operators applied to these chromosomes whose role is to create new members (children) in the population by crossing the genes of two chromosomes (crossover operators) or by modifying the genes of one chromosome (mutation operators).

6 **Replacement:** Natural selection of the members of the population who will survive.

7 **Parameter selections:** Natural convergence of the whole population that is globally improved at each step of algorithm.

The performance of a GA depends largely on the design of the above components and the choice of parameters such as population size, probabilities of genetic operators (crossover rate and mutation rate), and number of generations.

## 4.4 Near-To-Exact Algorithms

Near-to-exact algorithm give not exact but approach to best possible solution among all. These algorithm are also used to study and try to solve regarding to NP-hard.

Basically there are two types of algorithms for obtaining near-to-exact scheduling [14, 25].

(i) Approximation algorithms: These algorithms provide a theoretical guarantee for the quality of the obtained solution.

(ii) Heuristic algorithms: No such theoretical guarantee can be given. The quality of solution is determined by simulation experiments and actual implementation.

The performance of approximation is measured by approximation ratio, i.e., a function of size of input instance. Let $A$ be an algorithm then for any input instance of size $n$, $A$ has an approximation ratio of $\rho(n)$ if the cost $C$ of the solution produced by the algorithm is within a factor $\rho(n)$ of the cost $C*$ of the optimal solution, i.e., $\max\left(\frac{C}{C*}, \frac{C*}{C}\right) \le \rho(n)$ [7]. Regarding to online version al well as other optimization problems, the concept of competitive ratio is also introduced. Let $A_0$ be an online algorithm. For any instance of size $n$, let $C_A$ be the cost of solution obtained by $A_0$ and $C$ be the cost of optimal solution for the corresponding offline problem. Then $A_0$ is said have a competitive ratio of $\rho_c(n)$ if $C_A \le \rho_c(n)C$ [31].

### 4.4.1 Approximation Algorithms for Off-line Problems

Approximation technique is not a general paradigm. Depending upon the problem, one has to implement his own scheme for obtaining the solution. For example, consider the NP-hard problem $1|r_j|\sum F_j$. Keller et.al, obtained an approximation algorithm for this problem with an

approximation ratio of $O(\sqrt{n})$ for this problem. Their technique does not fall on any broad class of algorithms; the summary given as:

(i) Convert the problem $1|r_j|\sum F_j$ to $1|r_j; pmpt|\sum F_j$ by allowing preemptions,

(ii) Solve $1|r_j; pmpt|\sum F_j$ using Shortest Remaining Processing Time rule [5],

(iii) From the solution of preemptive version, obtain the solution for the original problem $1|r_j|\sum F_j$.

For the last step, they associate a forest structure for the preemptive schedule, such that each node represents an interval $[S_i, C_i]$ where $S_i$ and $C_i$ are the start and complete times of a job $J_i$ in the preemptive schedule. The solution for the original non-preemptive problem is obtained by merging these trees in a suitable way.

Savelsbergh et.al, make an empirical analysis of several approximation algorithms based on linear programming formulation for the problem $1|r_j|\sum w_j C_j$. They conclude that these techniques usually have complexity of $O(n \log n)$, $n$ being the number of jobs, and have very reasonable approximation ratio.

### 4.4.2 Approximation Algorithms for On-line Problems

On-line algorithms are betting larger attention by researchers of the scheduling theory. Consider the problem $1|r_j|\sum C_j$ even the off-line version of this problem is NP-hard [30]. Regarding its on-line version, some popular approaches for solving it are the FCFS (First Come First Serve) and SPT (Shortest Processing Time) rules. For the problem $1|r_j|\sum C_j$, both FCFS and SPT rules have competitive ratio of $n$, where $n$ is the total number of jobs that is a very pessimistic result. Again for the same problem, D-SPT (Delayed Shortest Processing Time) rule gives the competitive ratio of 2, which is a vast improvement compared to the performance of FCFS and SPT. The main idea behind D-SPT rule is to postpone a job with too large processing requirement.

D-SPT rule:

If the machine is idle and a job is available at time t, determine an unscheduled jobs with smallest processing requirement, say $J_i$. If there is a choice, take the job with smallest release

date. If $p_i \le t$, then schedule $J_i$, otherwise wait until time $p_i$, or a new arrives. Hoogeven et.al further proved that there can be no off-line algorithm for the problem $1|r_j|\sum C_j$ having competitive ratio less than 2. Their result was generalized by Anderson et.al, who proved that the D-WSPT (Delayed Shortest Weighted Processing Time) rule has a competitive ratio of 2 for the more general problem $1|r_j|\sum w_j C_j$. D-SWPT is very much similar to D-SPT; the difference is due to the weight.

D-SWPT rule:

Suppose that the machine is available at time t. We choose from among the available job as a job $J_i$ with the lowest value of ratio $\frac{p_j}{w_j}$ to start at time t, otherwise, we do nothing until time $p_j$ or another job is released if this occurs before time $p_j$. There can be no algorithm for the on-line version of the problem $1|r_j|\sum w_j C_j$ having competitive ratio 2 [31].

# CHAPTER 5

# SINGLE MACHINE SCHEDULING PROBLEM

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.

## 5.1 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.

It is worth-noting that when speaking about SMS problems two different perspectives are distinguished between them [33].

One is the category of lot sizing scheduling problems. In this type of problem, several different items 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 setup times occur each time the processed time is changed [34]. These problems aim at determining the optimal batch sizes to minimize cost under certain constraints. Bomberger (1966) mentions a metal stamping facility as an example. Stamps having different size and forms are produced on one press. Setup time and cost occur each time a forming die must be changed [34].

The order is the category focused on: lot sizes are fixed and aim is to determine a schedule satisfying when each job is to be executed in order to achieve certain objective. 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 given by [19, 33].

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 calls them building block in the development of a comprehensive understanding of scheduling concepts [19].

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 multiprocessor 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 appropriate especially when handling a small production unit to treat a complete production line on an aggregated single resource.

Focusing on more complex problems types with more than one machine in shop environment; 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 [14]. Single machine scheduling problem form the largest group within the area of scheduling research, consisting a variety of different settings and include a wider number of different constraints and objectives.

## 5.2 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*[45] |
| $[1\|\|\sum w_j C_j$ | O(nlogn) | W SPT-rule | *Smith*[45] |

| | | | |
|---|---|---|---|
| $[1\|\|F_{max}$ | | 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*[46] |
| $[1\|\|\sum w_jL_j)$ | 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*[46] |
| $[1\|\|\sum T_j]$ | NP-Hard | | |
| $[1\|\|\sum w_jT_j$ | NP-Hard | | *Lawler*[38] |
| $[1\|\|\sum U_j$ | O(nlogn) | Hodgson Algorithm | *Moore*[38] |
| $[1\|\|\sum r_jU_j$ | NP-Hard | | |
| $[1\|\|w_jU_j$ | NP-Hard | | *Karp*[37] |

Table 5.1: Complexity of elementary SMS problems

Table 5.1 shows overview - including the research results and directions for the main classes - is given by *Gupta and Kyparisis* [33]. 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.

## 5.3 Some SMS Related Problem Types

1. Minimizing maximum make-span [1| |Cmax], maximum flow time [1| |Fmax]: A minimization of the make-span equals minimizing idle time of the machine. Thus, each schedule that causes zero idle times is optimal. Since, flow time is defined as $F_j$ = Cj- $r_j$ , both problems are same if $r_j$ = 0 for all $J_j$.

   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| |$\sum L_j$]: Consider the general ease of Mi machines, where i = 1, 2,….,m, a decomposition of flow time leads to following relationships, [35]:

   $$F_j = \sum_{i=1}^{m} (w_{ji} + p_{ji}) = \text{Wj} + \sum_{i=1}^{m} p_{ji} \text{ and } \text{Fj} = \text{Cj} - r_j = d_j + \text{Lj} - r_j .$$

   Hence; $\sum$ Cj, $\sum$ Fj, $\sum$ Wj and $\sum$ Lj are equivalent.

   Objective functions since each can be modified to one of the others through linear transformation [35].

2. Minimizing total weighted completion time [1| |$\sum w_j$ Cj], total weighted flow time [1| |$\sum w_j$ Fj], total weighted waiting time [1| |$\sum w_j$ Wj], and total weighted lateness [1| |$\sum w_j$ Lj] : With use of relationship shown above, the equivalence of the objectives $\sum w_j$ Cj, $\sum w_j$ Fj, $\sum w_j$ Wj and $\sum w_j$ Lj becomes obvious [35].

   $$\sum w_j \text{Fj} = \sum w_j \text{Wj} + \sum_{i=1}^{n} w_j + \sum_{i=1}^{m} p_{ji}$$

   and $$\sum w_j \text{Fj} = \sum w_j \text{Cj} - \sum_{i=1}^{n} w_j r_j = \sum w_j \text{Lj} + \sum_{i=1}^{n} w_j (d_j - r_j).$$

Smith (1956) proved optimality of the WSPT rule for the total weighted completed time and hence for the other problems. This method is bounded polynomially by O(n log n).

3. Minimizing maximum waiting time [1| |Wmax]: It can be solved in O(n) by

simply scheduling the job with largest processing time last.

4. Minimizing maximum tardiness [$1||T_{max}$] and maximum lateness [$1||L_{max}$] : Jackson (1955) applied the EDD rule to solve problems with computational effort of O(n log n).

5. Minimizing total tardiness [$1||\sum T_j$] : The complexity of the problem of minimizing total tardiness remained open for a long time until [17] proved NP-hardness.

6. Minimization of total weighted tardiness [$1||\sum w_j T_j$]: It is the generalization of tardiness and showed as NP-hardness by Lawer and Lenstra et. al. (1977).
7. Minimizing the weighted number of tardy jobs showed NP-hardness [37] remaining even if all jobs have a common due date.

## 5.4 Polynomially solvable single machine scheduling problems:

Single machine scheduling is a classical scheduling problem. In this type of scheduling, n jobs are processed on one machine. Many problems are solved by single machine in polynomial time and these problems are known as easy problems. In this section, we will study about those problems that are solved by single machine in polynomial time.

**5.4.1** $1|prec|f_{max}$ **:**

To solve problem with $1|prec|f_{max}$ $f_{max} = \max_{j=1}^{n} f_j(C_j)$ and $f_j$ monotone for j = 1,2,….,n, it is sufficient to construct an optimal sequence $\pi : \pi(1), \pi(2), ....., \pi(n)$. Lawer developed a simple algorithm which developed a simple algorithm which constructs this sequence in a reverse order.

Let, N = {1, 2,…., n} be the set of all jobs and denoted by $S \subseteq N$ the set of unscheduled jobs. Furthermore, define p(S) $= \sum_{j \in S} p_j$. Then, the scheduling rule may be formulated as follows:

Schedule a job $j \in S$ which has no successor in S and has a minimal $f_j(p(S))$ value as the last job in S.

To give a precise description of the algorithm, represent the algorithm, represent the precedence constraints by corresponding adjacently matrix A = $(a_{ij})$ where, $a_{ij}$ = 1if and only if j is a direct successor of i. By $n(i)$, denote the immediate successor of i.

**Algorithm 5.1** [5] Lawer's algorithm for $1|prec|f_{max}$.

*begin*

1. *for i = 1 to n do n(i) =* $\sum_{j=1}^{n} a_{ij}$
2. *S = { 1,2,...,n}, p =* $\sum_{j=1}^{n} p_j$
3. *for k = n down to 1 do*

   *Begin*

   *3.1 jobs* $j \in S$ *with n(j) = 0 and minimal* $f_j(p)$ *value.*

   *3.2* $S = S|\{j\}$

   *3.3* $n(j) = \infty$

   *3.4 find* $\pi(k) = j$

   *3.5* $P = p\text{-}p_j$

   *3.6 for I = 1 to n do*

   *3.7 If* $a_{ij} = 1$ *then* $i = n(i) - 1$

   *end*

*end of algorithm*

The complexity of this Algorithm is $O(n^2)$.

**5.4.2** $1|\ |\sum U_j$

To generate an optimal schedule, it is sufficient to construct a maximal set of jobs which are on time. The optimal schedule then consists of the sequence of jobs in F ordered according to non-decreasing $d_j$-values followed by the late jobs in any order.

An optimal set F is constructed by the following rule: add jobs to non-decreasing due dates. If the addition of job j results in this job is being completed after $d_j$, then a job in S with the largest processing time is marked to be late and removed from F. The following algorithm, in which t denotes the current schedule time, implements this rule.

### 5.4.3 $1||w_jC_j$

This problem can be solved using the weighted shortest processing time (WSPT) rule. The SWPT rule is to sort jobs in non-decreasing order of $p_j|w_j$. This WSPT rule produces an optimal solution for problem $1||\sum w_jC_j$. The optimality of SWPT rule can be proved as a sequence of a more general theorem due to Lawer [30]. Consider this problem that includes $1||\sum w_jC_j$ as a special case: Given a set N of n jobs and a real valued function f which assigns $f(\pi)$ to each permutation $\pi$ of the jobs, find a permutation $\pi^*$ such that $f(\pi^*) = \min \{ f(\pi) | \pi$ is a permutation of N$\}$. If some special cases one can find a transitive and a complete relation $<$ on the set of jobs N such that for nay two jobs $J_i$, $J_k \in$ N, and for any permutation of the form $\alpha J_i j_k \delta$,

$$J_i < J_k \Rightarrow f(\alpha J_i J_k \delta) < f(\alpha J_k J_i \delta) \qquad \text{.....(5.1)}$$

If such a relation exists for a given function $f$, $f$ is said to admit the relation $<$, and the relation $<$ is known as a task interchange relation for $f$. Now consider the following theorem:

## 5.5 Other Problems

$1|r_j|\sum U_j$:

The algorithm for this problem also constructs a minimal set S of jobs completed on time, which are scheduled in arbitrary order. Lawer devised an algorithm based on this idea and prove that it gives optimal solution for $1|r_j|\sum U_j$.

$1|\ |\sum U_j$ :

The algorithm for $1|\ |\sum U_j$ is very much similar to the algorithm for problem $1|p_j = 1|\sum w_j U_j$ . This problem can be solved in O(n logn) time using Moore's algorithm [40]. The algorithm constructs a minimal set S of jobs which complete on time. The optimal solution then consists of the jobs in S scheduled according to EDD rule, followed by the late jobs in any order. The set S is constructed by Moore's rule: add the jobs in S in order of non-decreasing due dates. If the addition of jobs $J_j$ results in this job being computed after $d_j$, then a job in S with the largest processing time is marked to be late and removed from S.

$1|pmtn; prec; r_j|f_{\max}$ :

The objective function $f_{\max}$ for this problem is $f_{\max}$ = max { $f_j(C_j)$ }. This problem can be solved in $O(n^2)$ time using the following algorithm with this steps:

If a job $J_j$ is a successor of a job $J_i$ and $r_i + p_i > r_j$ then job $J_j$ cannot start before $r_j' = r_i + p_i$. So, replace $r_j$ by $r_j'$. In this way, all release dates are modified.

Schedule the jobs in non-decreasing order of modified release dates. This decomposes the jobs into blocks, where a block in a minimal set of jobs processed without idle time between them.

Find optimal solution for each block separately. The resulting set of blocks will be the optimal schedule

$1|p_j = 1|\sum w_j U_j$

The objective function in this problem involves unit penalty $U_j$, this means for each job $J_j$ due dates $d_j$ is given here is an algorithm for $1|p_j = 1|\sum w_j U_j$ is described. This algorithm constructs an optimal set S of early jobs. To get an optimal schedule jobs in S are scheduled according to non decreasing due dates. Late jobs i.e. the jobs not belonging to S are scheduled in arbitrary order.

The main strategy of this algorithm is to construct the set S of early jobs such that total weight of jobs in S is maximum. For this one tries to schedule the jobs in earliest due date order. If a job i

to be scheduled next is late, then i is scheduled and a job k with smallest $w_k$ value is removed from S.

In the following algorithm t denotes the current time, n is the total number of jobs, and assume jobs are enumerated such that $1 \le d_1 \le \ldots\ldots\ldots \le d_n$.

**Algorithm 5.2** $1|p_j = 1|\sum w_j U_j$

*Begin*

*1 t=1, S= Φ*

*2 for i=1 to n do*

*3 if $d_i \ge t$ then*

*4 add i to S, t=t+1.*

*5 If there exist a job k with $w_k < w_i$*

*begin*

*6 Delete job k from S where k is the largest index such that $w_k$ is minimal.*

*7 Add i to S.*

*end*

*End of algorithm.*

If the scheduled jobs in S are organized as a priority queue with respect to $w_i$ value, the complexity of this algorithm is O (nlogn).

**Theorem5.1** [5] Algorithm $1|p_j = 1|\sum w_j U_j$ provides an optimal schedule.

Proof: Let s be sequence of jobs scheduled early by the algorithm ordered according to their indices. Let $S^*$ be the corresponding sequence of an optimal schedule coinciding with S as long as possible. Let k be the first job in $S^*$ which does not belong to. When constructing S, job k must have been eliminated by some job say i. let J be the set of jobs in S between k and I at the time k was eliminated. Due to step 6 of the algorithm $w_j < w_k$ for all j J. Thus all j must belong to

$S^*$, otherwise replacing k by j would yield a better schedule than $S^*$. However this implies that there is a late job in $S^*$ which is a contradiction.

**Example 5.1:** To demonstrate how the algorithm for $1|p_j=1|\sum w_jU_j$ constructs the set of early jobs consider jobs $J_1$,$J_2$ ,$J_3$,$J_4$,$J_5$, with the following information.

| j | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $W_j$ | 3 | 5 | 1 | 4 | 4 |
| $d_j$ | 3 | 6 | 2 | 8 | 2 |

Initially t=1 and the set of early jobs S= Φ.

At i=1, $d_1 \geq t$, so S= {$J_1$}, t:=t+1=2.

At i=2, $d_2 \geq t$, so S= {$J_1$,$J_2$}, t:=t+1=3.

At i=3, $d_3 < t$, $J_2$ has maximum weight in S, so swap $J_2$ and $J_3$ and S={$J_1$,$J_3$}.

At i=4, $d_4 \geq t$, so S={$J_1$, $J_3$, $J_4$}, t:=t+1=4.

At i=5,$d_5 < t$, $J_4$ has maximum weight in S, so swap $J_4$ and $J_5$, and S={$J_1$, $J_3$, $J_5$} finally.

The jobs in S are to be scheduled as per the sequence =($J_1$, $J_5$, $J_1$), $J_2$ and $J_4$ can be scheduled afterwards in any order.

Finally, the job in S are to be scheduled as per the sequence π={$J_3$, $J_5$, $J_1$} $J_2$ and $J_3$ can be scheduled afterwards in any order.

## 5.6 NP-hard Problem Related to SMP

### 5.6.1 $1|r_j|L_{max}$

A generalization of $1||L_{max}$ is the problem $1|r_j|L_{max}$ with the jobs released at different points in the time. It does not allow preemption and is significantly harder than the problem with all jobs available at time zero. The optimal schedule is not necessarily a non-delay schedule. It may be advantageous to keep the idle just before the release of a new job.

**5.6.21** $|w_jU_j$

This problem is also known to be NP-hard. The special case with all due dates being equal is equal and equivalent to so called Knapsack problem. The due date is equivalent to the size of Knapsack, the processing time of the jobs are equivalent to the benefits obtained by putting the items into the Knapsack. A popular heuristic for this problem is the WSPT rule which sequences the jobs in decreasing order of $\frac{w_j}{p_j}$. A worst case analysis shown that this heuristic may perform arbitrarily.

**Example 5.2** (WSPT rule and Knapsack):

Consider the following three jobs

| Jobs | 1 | 2 | 3 |
|---|---|---|---|
| $p_j$ | 11 | 9 | 90 |
| $w_j$ | 12 | 12 | 89 |
| $d_j$ | 100 | 100 | 100 |

Comparison to Schedule Jobs According to WSPT and Knapsack Rules.

Scheduling the jobs according to WSPT results if the schedule $1 \rightarrow 2 \rightarrow 3$. The third job is completed late and $\sum w_jU_j$ (WSPT) is 89. Scheduling the jobs according to $2 \rightarrow 3 \rightarrow 1$ results in $\sum w_jU_j$ (OPT) being equal to 12.

# CHAPTER 6

# MINIMIZING THE WEIGHTED NUMBER OF TARDY JOBS

## 6.1 Problem Presentation

This section addresses the problem $[1||\sum W_jU_j]$ i.e. minimizing the weighted number of tardy jobs on a single machine subject to certain job characteristics, n jobs, j = 1, ……., n (alternatively $J_j = J_1, \ldots.., J_n$), are to be schedule single machine. The job's processing time is $p_j$, its release date $r_j$ (i.e $r_j=0$) and its due date $d_j$, 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. Specific weight ($w_j$) are assigned representing a job's penalty incurred in case of tardiness. The objective is to minimize the weighted number of tardy jobs; or in other words to minimize the penalty payments $\sum W_jU_j$.

A property often referred to shall be prefixed is that if a job j is tardy it might as well be arbitrary 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 weighted number of tardy jobs" is equivalent to "maximizing the weighted number of on-time jobs".

This section is organized as following.

Starting with Section 6.2, a detailed literature review is given.

Further, Section 6.3 addresses some "previous" solution approaches.

## 6.2 Historical Development and Research Overview

A comprehensive up to date overview of the research effort limited to the problem specification mentioned above, namely those without release date and preemption is not given. Allowing for the vast field of publications and research made, it is hardly possible to guarantee completeness of such an overview. Thus, we rather aim at pointing at the important milestones and may be a

little more which enable orientation and impetus for interested reader, instead of claiming completeness.

NP-hardness of $[1||\sum W_jU_j]$ was first proven by Karp [37] and holds even if all jobs have a common due date. The publication of Lawler and Moore [38] is seen as pioneer work providing a dynamic programming approach pseudopolynomially bounded by $(n \min \{\sum_j p_j, \max_j \{d_j\}\})$. Their formulation was generalized by Sahani [39]. Under the assumption that all weights have to integers the approach is pseudopolynomially bounded by $O(n \min \{\sum_j p_j, \sum_j w_j, \max_j \{d_j\}\})$ [41].

The drawback of both formulations is the complexity's dependence on the input data, which limit their practical use. It was again Lawler [40] who adopted the well known Moore Hodgson algorithm to easily solve the weighted problem as special case if processing time and weights can be indexed such that they are oppositely ordered, i.e. $p_i < p_j \Rightarrow w_i \geq w_j$ for every i,j. The time bound is $O(n \log n)$.

Furthermore, Lawler [23] presented a new dynamic programming approach, initially for the case with preemption and release time constraints, that solves $[1||\sum W_jU_j]$ as a special case in $O(nW)$, with W denoting the sum of integer job weights.

Considering the branch and bound algorithm,[41] proposed a procedure solving problems with up to 50 jobs; the procedure of Tang [42] is able to solve up to 85 jobs. Potts and Van Wassenhove [15] proposed a branch and bound method appropriate for up to100 jobs; and as one of the newest publications M'Hallah and Bulfin [43] is worth mentioning coping up to 2500 jobs.

When adding release dates to the problem, i.e. $[1|r_j|\sum w_jU_j]$, Lenstra et al. [24] showed that it remains to be NP-hard, even for the case with unity weights. Techniques applied to this type of problem cover a wide field. The dynamic programming approach by Lawler [44] mentioned above, initially for the preemptive case, solves the non-preemptive case under the additional assumption that release dates and due dates are similarly ordered, i.e. $r_i < r_j \Rightarrow d_i \leq d_j$ for every i,j, on $O(nW)$.

If the processing times are equal ($p_j=p$ for every j) the dynamic programming formulation of Baptise [3] achieves attention solving the problem in polynomially bounded time of $O(n^7)$. Some newer branch and bound algorithm methods were proposed by Peridy et al. (2003) and

M'Hallah and Bulfin (2007). The first utilizes a Lagrangean based lower bound, the latter surrogate relaxation that leads to a multiple choice knapsack formulation to compute the bounds.

A hybrid branch and cut method with improved infeasibility cuts based on constraint propagation is suggested by Sadykov (2004). Further, a heuristic developed by Dauzere-Peres and Sevaux (2003) is worth mentioning, computing a lower bound based on the notion of a so called master sequence and Lagrangean relaxation, as well as a genetic algorithm by Sevaux and Dauzere-Peres (2003).

The last problem type investigated that is $[1|pmtn,r_j|\sum w_jU_j]$ is NP-hard as well, but it can be solved in pseudopolynomial time by the dynamic programming algorithm of Lawler (1990), bounded by O ($nk^2W^2$), with W as the sum of integer weights and k as the number of distinct release dates. For unity processing times, i.e. $p_j=p$ for every j, Baptiste (1990b) developed an O ($n^{10}$) dynamic programming formulation. If the processing times and release dates can be indexed such that a similar order is obtained and processing times and weights are oppositely ordered on the other hand ($p_i < p_j \Rightarrow r_i \leq r_j$ and $p_i < p_j \Rightarrow w_i \geq w_j$ for every i,j) or if release date and due date intervals are nested and processing times and job weights oppositely ordered, Lawler (1994) suggests both underlying dynamic programming formulations to be solved by variation of the Moore-Hodgson algorithm in O ($n\log n$).

## 6.3 Similar 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.

| Weighted Number of Tardy Jobs $[[1|\ldots\ldots|\sum w_jU_j]$ | | | | | | |
|---|---|---|---|---|---|---|
| Specified Problem | Year | Author | Solving Method | Character | Complexity | Comment |
| $[1\|\|\sum w_jU_j$ | 1969 | *Moore and Lawler* | DP | En | O(n min $\{\sum_j p_j, \max_j\{d_j\}\}$)) | |
| $[1\|\|\sum w_jU_j]$ | 1976 | *Sahni* | DP | En | O(n min $\{\sum_j p_j, \sum w_j, \max_j\{d_j\}\}$) | All weights integers |

| | | | | | | |
|---|---|---|---|---|---|---|
| $[1\|\|\sum w_jU_j]$ | 1976 | *Lawler* | Generalization of Moore algorithm | Op | O(nlogn) | $p_i<p_j \Rightarrow w_i \geq w_j$ |
| $[1\|\|\sum w_jU_j]$ | 1983 | *Villareal & Bulfin* | B &B | En | | For up to 50 jobs |
| $[1\|\|\sum w_jU_j]$ | 1988 | *Potts and Van Wassenhove* | B & B | En | O(nlogn) | For up to 100 jobs |
| $[1\|\|\sum w_jU_j]$ | 1990 | *Tang* | B & B | En | | For up to 85 jobs |
| $[1\|\|\sum w_jU_j]$ | 1990 | *Lawler* | DP | En | O(nW) with W as sum of integer job weights | All weights integers |
| $[1\|\|\sum w_jU_j]$ | 2003 | M'Hallah & Bulfin | B & B | en/he | | For up to 2500 jobs |
| $[1\|r_j\|\sum w_jU_j]$ | 1990 | *Lawler* | DP | En | O(nW) with W as sum of integer job weights | All weights integers $r_i<r_j \Rightarrow d_i \leq d_j$ |
| $[1\|r_j,p_j=p\|\sum w_jU_j]$ | 1999 | *Baptiste* | DP | En | $O(n^7)$ | Equal processing times $p_j=p$ |
| $[1\|r_j\|\sum w_jU_j]$ | 2003 | *Dauzere-Peres & Sevaux* | Heuristic (Lagrangean Relaxation) | He | | For more than 100 jobs |
| $[1\|r_j\|\sum w_jU_j]$ | 2003 | *Dauzere-Peres & Sevaux* | Genetic algorithm | He | | |
| $[1\|r_j\|\sum w_jU_j]$ | 2003 | *Peridy et al* | B&B | En | | |
| $[1\|r_j\|\sum w_jU_j]$ | 2004 | *Sadykov* | Branch &cut | En | | |
| $[1\|r_j\|\sum w_jU_j]$ | 2007 | *M'Hallah and Bulfin* | B&B | en/he | | |
| $[1\|pmtn,r_j\|\sum w_jU_j]$ | 1990 | *Lawler* | DP | En | $O(nk^2W^2)$ with k as number of distinct release dates & W as sum of weights | All weights integers |

| | | | | | | |
|---|---|---|---|---|---|---|
| $[1\|pmtn,r_j\|\sum w_jU_j]$ | 1994 | *Lawler* | Generalization of Moore's Algorithm | Op | O(nlogn) | $p_i<p_j \Rightarrow r_i \leq r_j$ $p_i<p_j \Rightarrow w_i \geq w_j$ |
| $[1\|pmtn,r_j\|\sum w_jU_j]$ | 1994 | *Lawler* | Generalization of Moore's Algorithm | Op | O(nlogn) | Nested release date –due date intervals $p_i<p_j \Rightarrow w_i \geq w_j$ |
| $[1\|pmtn,r_j,p_j=p\|\sum w_jU_j]$ | 1999 | *Baptiste* | DP | en | $O(n^{10})$ | Equal processing times: $p_j = p$ |
| DP=Dynamic Programming B&B=Branch and Bound ,op=optimal ,en=enumerative, he=heuristic | | | | | | |

# CHAPTER 7

# PROBLEM STATEMENT AND METHODOLOGY

In the last chapter we discussed the research done on minimizing the weighted 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.

## 7.1 Statement of the Problem

The problem in the present research is, scheduling of n jobs that are available at time $r_j$ (i.e. $r_j$=0) on a single machine to minimize the weighted number of tardy jobs. Each job is associated with a release time constant, 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.

## 7.2 Assumptions

1. The machine is always available

2. No preemption is allowed.

## 7.3 Objective Function

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

Minimize$\sum_{WjUj}$, for j = 1, 2,.., n

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

## 7.4Complexity of the Problem

The 1 | |$\sum_{WjUj}$, is proved to be a NP-hard problem and an optimal solution to this problem can be found only in pseudo-polynomial time. I implement a dynamic programming algorithm and branch and bound algorithm for the above problem and obtained result from both are compared.

.

## 7.5 Methodology

### 7.5.1 A Branch and bound approach

A B&B approach by M'Hallah and Bulfin (2003) for a problem with equaling release dates ($1||\sum w_jU_j$) is treated[46]. In table 5.1 shows, many very different B&B approaches for minimizing the weighted number of tardy jobs on one singe machine have been proposed by several authors. The very special approach by is discussed in detail since it is capable of solving typical problems with up to 2500 instances in about twelve minutes[46]. The approach falls back upon a heuristic that provides good solution on instances with up to 2500 jobs in a few seconds. Another particular feature is the utilization of a mathematical programming formulation based on the well known knapsack problem to develop a bound. Actually, the resulting algorithm can be implemented with slight modifications to any knapsack code. In order to keep the algorithm as simple as possible, neither dominance nor reduction properties are embedded.

#### 7.5.1.1 Derivation of a knapsack model

Instead of minimizing the weighted number of tardy jobs, it is possible to seek for a maximum weight feasible set of on time jobs. The original problem referred to as **WNT** is formulated as following, assuming the jobs to be indexed in non decreasing order of their due dates.

$$\max \sum_{j=1}^{n} w_j x_j \qquad \text{(WNT)}\ldots\ldots\ldots\ldots\ldots\ldots\ldots(7.1)$$

$$\text{s.t } \sum_{i=1}^{j} p_i x_i \leq d_j \qquad j=1, \ldots.n,\ldots\ldots\ldots\ldots\ldots\ldots(7.2)$$

$$x_j \in \{0,1\} \qquad j=1, \ldots n, \ldots\ldots\ldots\ldots\ldots\ldots(7.3)$$

With $x_j$ equaling 1 if job j is scheduled on time and 0 otherwise, the objective function equation (7.1) simply sums the weights of all on-time jobs. The set of constraints in equation (7.2) ensures that any on-time job's completion time is smaller or equal than its due date in consideration of the on times job's with smaller due dates sequenced before. The final set of constraints in (7.3) ensures that each job is either scheduled on time or tardy.

Surrogate multipliers are used for obtaining the well known knapsack model as a relaxation of WNT. Glover was the first who introduced surrogate constraint to integer programming. Generally, surrogate relaxation subsumes several constraints by weighting them with factors so called surrogate multipliers, and by summing up these modified constraints [43]. With $\lambda_i \geq 0$ being the surrogate multiplier associated with constraint i and with

$$a_j = p_j \sum_{i=j}^{n} \lambda j \text{ and } b = \sum_{j=1}^{n} \lambda_j d_j$$

The knapsack model(referred to as KP) can be expressed as following

$$\max \sum_{j=1}^{n} w_j x_j \qquad \text{(KP) ……..(7.4)}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} aj x_j \leq b \qquad \text{……... (7.5)}$$

$$x_j \in \{0,1\} \qquad j=1, \ldots . , n \text{ ……... (7.6)}$$

KP corresponds to the binary knapsack problem, with $w_j$ being the profit if item j is selected ($x_j=1$; otherwise $x_j=0$), $a_j$ being the weight of item j and b being the knapsack capacity. The objective of the profit equation is simply expressed as the sum of all packed items profit, where as equation ensures that the overall weight does not exceed the knapsack capacity.

Any feasible solution to WNT is also feasible to KP, but not necessarily the other way round. Thus, KP is less constraint providing a bound for WNT. The bound's quality, of course depends on the surrogate multipliers values. The linear programming of KP (i.e LPKP) obtained by relaxing the binary restriction in (7.6) and allowing $x_j$ to take real values between zero and one (i.e. $0 \leq x_j \leq 1, j=1,, \ldots, n$ ) can easily solved. After reordering the variables in decreasing order of $w_j/a_j$, i.e.

$$w_1/a_1 \geq w_2/a_2 \geq \ldots\ldots \geq w_n/a_n,$$

One has to find the index f satisfying

$$\sum_{i=1}^{n} ai \leq b \leq \sum_{i=1}^{f+1} ai$$

Then, in an optimal solution, all jobs whose index is smaller than f-1 are set on time, while jobs associated with an index larger than f+1 are scheduled tardy. If job j cannot be scheduled

completely it is scheduled partially. Thus an optimal solution of LPKP has at most one variable with $0<x_j<1$. Subsuming, an optimal solution of LPKP with identified index f has the following form.

$x_j=1$, $j=1,\ldots\ldots, f-1, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(7.7)$

$x_j = b-\sum_{i=1}^{f} ai/a_f$ $j=f \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(7.8)$

$x_j=0$ $j=f+1, \ldots, n, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(7.9)$

An O(n) algorithm for finding a critical job based on critical ratios stems from Balas and Zemel.

**7.5.1.2 A heuristics for obtaining an initial solution for WNT**

Given suited surrogate multipliers, the heuristic procedure by M'Hallah and Bulfin runs in O ($n^2$) polynomial time. It is used to find the initial feasible solution for WNT as starting point for the exact B&B algorithm.

The heuristic starts by solving LPKP and checking whether all jobs in O, which denotes the subset of those supposed to be on time, are actually on time if scheduled in EDD order (the EDD schedule is said to be feasible) (step 0 and 1).Note that all jobs with an index smaller or equal than f-1 are initially supposed to be on time, Consequently, j=1,….., f-1 and O={1,….., f-1}. If in EDD ordering not all such jobs (in O) can actually be scheduled on time (the solution to LPKP is not feasible to WNT), the last job in O (at the beginning the one with index f-1) is removed and set tardy ($x_{f-1}=0$), the resulting EDD schedule is checked again for feasibility. This procedure is continued (step 2) until a feasible EDD schedule to WNT is found. The heuristic then tries to improve it by gradually setting tardy jobs temporarily on time (adding them to O) (step3) and by testing the schedule's feasibility (step 4). If infeasibility is triggered again, the last job which was temporarily set on time is set tardy again. As the consequence of the LPKP solving procedure the sequence in which jobs are considered depends on $w_j/a_j$, i.e. the job's ratio of objective coefficient ($w_j$) to constraint coefficient ($a_j$). On time jobs are scheduled tardy, one by one, in increasing order of $w_j/a_j$ whereas tardy jobs are scheduled on time in decreasing order of $w_j/aj$

**Algorithm 7.1 [46] Heuristic**

Step 0: Solve LPKP; set O= {1,2, ….., f-1}, L={f, f+1, ….., n},k=f-1.

Step1: Schedule all jobs in O in EDD-order. If all jobs in O are on time, the EDD schedule is feasible; proceed with step 3.

Step 2: Set O=O-{k}, L=LU{k} and k=k+1. Go back to step 1.

Step 3: If L= go to step 5 otherwise set k=k+1, L=L-{k}.

Step 4: If all jobs in OU{k} are on time in EDD order set O= OU{k} and go back to step 3.

Step 5: All jobs in O are on time. The objective value of WNT equals:

$Z^*=\sum_{j\epsilon O} wj$ and $x_j^*=1$,j ϵ O; $x_j^*=0$ otherwise.

**7.5.1.3 An exact algorithm for WNT**

Starting with initial solution for WNT obtained by applying the heuristic discussed in the previous section, the B&B approach uses bounds based on LPKP. Each problem in the candidate list (each node) is a subproblem of KP having certain variables fixed to either one or zero. At each iteration, the subproblem with the best bound (the candidate problem CP) is chosen from the candidate list and its linear programming relaxation (LPKP) is solved, thus maximum upper bound rule is applied. When no feasible solution can be found, or the objective function's value is lower/ equal than the best known feasible solution, the branch can be fathomed and another problem is chosen from the candidate list. Otherwise if the solution obtained by solving LPKP is better than the best known feasible solution, it is checked whether this solution is also feasible to WNT. According to the set of constraint (7.2) and (7.3) all jobs thought to be on time actually must be in on time on a EDD schedule, and all variables must be integer. If all variables are integer and if all jobs j with $x_j=1$ are on time in an EDD schedule, anew best solution is found. Problems with bounds no better than this new best solution's objective value are removed from the candidate list otherwise some variable are fixed, the treated candidate problem is decomposed and two new candidate problems are added to the candidate list .

### 7.5.2 Dynamic programming applied to problem.

Given $n$ jobs $i=1,2,.....,n$ with processing times $p_i$ and due dates $d_i$. These jobs have to be sequenced such that $\sum_{i=1}^{n} w_i U_i$ is minimized where $w_i \geq 0$ for $i=1,2,.....,n$. Assume that the jobs are enumerated according to non-decreasing due dates: $d_1 \leq d_2 \leq ..... \leq d_n$. Then, there exists an optimal schedule given by a sequence of the form, $i_1, i_2, ....., i_s, i_{s+1}, ...., i_n$. Where, jobs $i_1 < i_2 < ..... < i_s$ are on-time and jobs $i_{s+1}, ...., i_n$ are late. If a job $i$ is late, then put $i$ at the end of the schedule without increasing objective function. If $i$ and $j$ are early jobs with $d_i \leq d_j$ such that $i$ is not scheduled before $j$ then shift the block of all jobs scheduled between $j$ and $i$ jointly with $i$ to the left by $p_i$ time units and schedule $j$ immediately after this block. Since, $i$ was not late and $d_i \leq d_j$ which creates no late jobs. To solve the problem, calculate recursively for $t=1,2,.....,T=\sum_{i=1}^{n} p_i$ and $j=1,2,.....,n$ the minimum criterion value $F_j(t)$ for the first $j$ jobs subjected to constraint that total processing time of on-time job is at most $t$. If $0 \leq t \leq d_j$ and job $j$ is on-time in a schedule corresponds with $F_j(t)$ than $F_j(t) = F_{j-1}(t - p_j)$. Otherwise $F_j(t) = F_{j-1}(t) + w_j$. If $t > d_j$, then $F_j(t) = F_j(d_j)$ because all jobs $1,2,....., j$ finishing later than $d_j \geq ..... \geq d_1$ are late.

To calculate an optimal schedule it is sufficient to calculate the set $L$ of late jobs in an optimal schedule.

**Algorithm 7.2**: $1 \mid \mid \sum w_i U_i$

1. *for* $t = -p_{\max}$ *to* $-1$
2. *for* $j = 0$ *to* $n$

   $F_0(t) = \infty;$
3. *for* $t = 0$ *to* $T$

   $F_0(t) = o;$
4. *for* $j = 1$ *to* $n$

   {

*4.1 for* $t = 0$ *to* $d_j$

*4.2 if* $F_{j-1}(t) + w_j < F_{j-1}(t - p_j)$

$F_j(t) = F_{j-1}(t) + w_j \; F_j(t)$ ;

*else*

$F_j(t) = F_{j-1}(t - p_j)$ ;

*4.3 for* $t = d_j + 1$ *to* $T$

$F_j(t) = F_j(d_j)$ ;

*}*

### 7.6 Experiments and Results

In this experimentation part, all of the algorithms mentioned in this chapter were implemented in java. The source codes for these programs are given in appendix. These algorithms were executed in Intel Pentium IV processor, Windows XP operating system, 512 MB of RAM. The programming language used was java.

The objective of implementing dynamic programming and branch and bound algorithms describe in this chapter, is simply to compare their output .

#### 7.6.1 Input Data Set

First, a program for generating data was implemented (see appendix). This program uses the random number generator provided by the java library. Using this program, size sets of input data were generated, containing instances for 10, 20, 30, 40 and 50 jobs. Each input data set contains 50 instances. In all instances, processing times are in range [1, 100]; due-dates are in range [1, 100] and weights for each job are in range [1, 10]. The release time for each job is set to be zero.

### 7.6.2 Output

| Input data set | No. of jobs | Weighted Number of Tardy Jobs(Dynamic Programming) | Weighted Number of Tardy Jobs (Branch &Bound) |
|---|---|---|---|
| 1 | 5 | 2 | 0 |
| 2 | 10 | 3 | 2 |
| 3 | 15 | 9 | 2 |
| 4 | 20 | 9 | 3 |
| 5 | 25 | 7 | 3 |
| 6 | 30 | 13 | 4 |
| 7 | 35 | 16 | 4 |
| 8 | 40 | 19 | 5 |

**Table 7.1**: Weighted number of tardy jobs given by various algorithms *(for all input data sets, no. of instances=50, maximum processing time=100, maximum due date=100, maximum weight for each job=10, and release time for each job=0).*

The above table is summarized in the following figure:

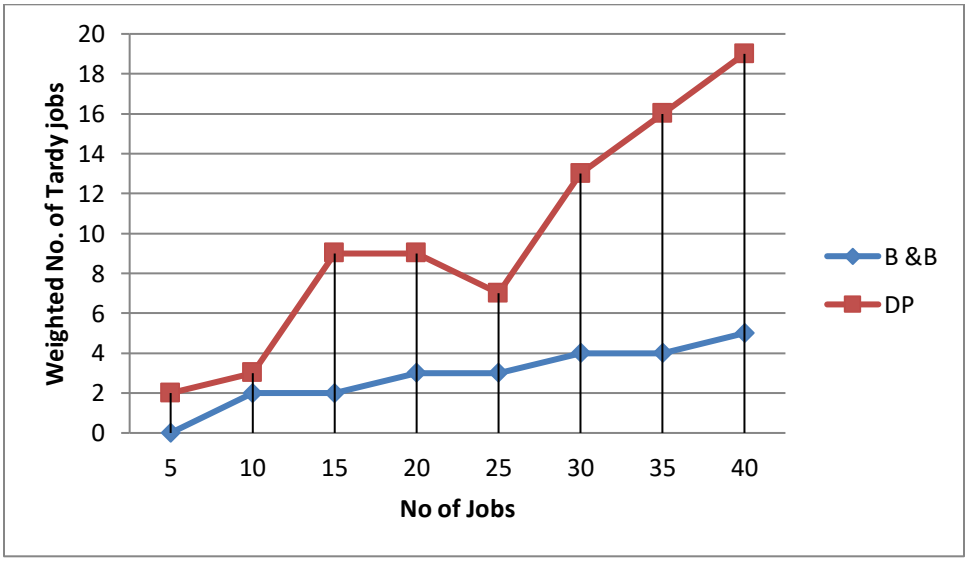



**Fig 7.1**: Weighted Number of Tardy Jobs Given By Various Algorithms.

From figure, it is clear that, Branch and bound algorithm is more efficient than dynamic programming.

# CHAPTER 8

# CONCLUSION AND RECOMMENDATION

In Chapter 8 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, I addressed problem related to minimize the weighted number of tardy jobs, a single machine scheduling problem with release time constant performance objective of minimizing the weighted number of tardy jobs.

The problem is proved to be NP-Hard problem, and only can be solved in pseudo polynomial time. I 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.

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

## 8.2 Conclusions

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

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

## 8.3 Recommendation

The scheduling problem discussed in this thesis is deterministic single machine scheduling problem with a performance objective of minimizing the weighted number of tardy jobs with release time constants 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.

# REFERENCES


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

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

[3]. 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).

[4]. Graham, R.E., Lawer, E.L., Lenstra. J.K., and Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling, a survey", Annals of Discrete Mathematics 5(1979) 287-326

[5]. Brucker, P. (1995): *Scheduling Algorithms*. Springer, Berlin

[6]. Wikipedia, The free Encyclipedia.

[7]. Cormen, T. H., Leisrrson, C. E., Rivest, R. L., and Stein, C. (2004): *Introduction to Algorithms*. Prentice-Hall of India Pvt. Ltd.

[8]. Hopcroft, J. E., Motwani, R., and Ullmann, J. D. (2002): *Introduction to Automata Theory, Languages and Computation*. Pearson Education.

[9]. Silberschatz, A., Galvin, P. B., and Gagne, G. (2002): *Operating System Concepts*.6th Edition, John-Wiley Pvt. Ltd., New York.

[10]. Brasel Heidemarie, *"Latin Rectangles in Scheduling theory a basic modeling concept of LISA, 1996*

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

[12]. McCarthy, B. L., and Liu, J. (1993): *Addressing the Gap in Scheduling Research: A Review of Optimization and Heuristic Methods in Production Scheduling*. International Journal of Production Research, Vol. 31, pp. 59–79.

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

[14]. Blazewicz, J. C., Ecker, K. H., Pesch, E., and Weglarz, J. (1996): *Scheduling Computer and Manufacturing Processes*. Springer, Berlin.

[15]. Miltenburg, J. and Sinnamon, G., "Scheduling mixed model multi-level just-in-time production systems", International Journal of Production Research 27,9 (1989) 1487-1509.

[16]. Monden, Y., "Toyota production system", Industrial Engineers and Management press, Norcross, GA(1983).

[17]. 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.

[18]. Dasgupta, S. C., Papadimitriou, C. H., and Vazirani, U. (2006): *Algorithms*. McGraw-Hill.

[19]. Baker, K. R. (1974): *Introduction to Sequencing and Scheduling*. John Wiley and Sons, New York.

[20]. Winston, W.L. and M. Venkataramanan (2003): Introduction to mathematical programming. 4th edition. Toronto: Books/Cole-Thomson Learning.

[21]. Lawler, E. L. and Wood, D. E. (1966): *Branch and Bound Methods: A Survey*. Operations Research, Vol. 14, pp. 699–719.

[22]. Karger, D., Stein, C. and Wein, J. (1997): *Scheduling Algorithms*. In: M. J. Atallah, editor: *Handbook of Algorithms and Theory of Computation*. Boca Raton: CRC Press

[23]. 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.

[24]. Lenstra, J. K., Rinnoy Kan, A. H. G., and Brucker, P. (1977): *Complexity of Machine Scheduling Problem*. Annals of Discrete Mathematics, Vol. 4, pp. 121-140.

[25]. Brassard, and Bartley, P.(1998): *Fundamentals of Algorithmic,* prentice-Hall of India Pvt,Ltd.

[26]. Gere Jr., W. S. (1966): *Heuristics in Job Shop Scheduling.* Management Science, Vol. 13, pp. 167–190.

[27]. Blackstone, J. H., Phillips, D. T., and Hogg, G. L. (1982): *A State-of-the-Art Survey of Dispatching Rules for Job Shop Operations*. International Journal of Production Research, Vol. 20, pp. 27–45.

[28]. Russell, S., and Norvig, P. (2006): *Artificial Intelligence: A Modern Approach*. Pearson Education

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

[30]. Holland, J. H. (1975): *Adaptation in Natural and Artificial Systems*. Michigan University Press, Ann Arbor.

[31] Anderson, E. J., and Potts, C. N. (2002): *On-Line Scheduling of a Single Machine to minimize Total Completion Time*. Proceedings of the Thirteenth Annul ACM-SIAM Symposium on Discrete Algorithms, SODA, January.

[32]. 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.

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

[34]. Bomberger, E. E. (1966): *A Dynamic Programming Approach to a Lot size Scheduling Problem*. Management Science, Vol. 12, pp. 778–784.

[35]. Domschke, W., Scholl, A., and Voß, S. (1993): *Produktionsplanung - Ablauforganisatorische Aspekte*. Springer, Berlin.

[36]. Domschke, W., and Drexel, A. (2005): *Einf¨uhrung in Operations Research*. 6th edition. Springer, Berlin.

[37]. Karp, R. M. (1972): *Reducibility Among Combinatorial Problems*. In: Miller, R. E., and Thatcher, J. W. (editors): *Complexity of Computer Computations*. Plenum Press, New York, pp. 85-103.

[38]. Lawler, E.L,and J.M. Moore (1969): A functional equation and its application to resource allocation and sequencing problems. *Management science,16, 77-84.*

[39]. Sahni,S. (1976): Algorithm for scheduling independent jobs. *Journal of the Association of Computing Machinery, 23, 116-127.*

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

[41]. Villareal, F.J. and R.L. Bulfin (1983): Scheduling a single machine to minimize the weighted number of tardy jobs. *IIE Tranctions, 15, 337-343.*

[42]. Tang, G. (1990): A new branch and bound algorithm for minimizing the weighted number of tardy jobs. *Annals of Operations Research, 24, 225-232.*

[43]. M'Hallah, R. and R. L. Bulfin (2003): Minimizing the weighted number of tardy jobs on a single machine. *European Journal of Operational Research,145, 45–56.*

[44]. 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.

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

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

# APPENDIX A: Program Source Code of Algorithms

```
package javaapplication3;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;
public class FindTardyJobs {
public static double num, num1;
   public static int n = 8;
   //array to hold processing times
   public static int p[] =  new int[n];

   //array to hold due times
   public static int d[] =  new int[n];

   //array to hold weight
   public static int w[] =  new int[n];

   public static int tardyJob = 0;

   //array to hold the optimal schedule given  by the sequence
   public static int onTimeLateJobs[] = new int[n];

   //arraylist to hold the ontime jobs
   public static ArrayList<Integer> onTime = new ArrayList<Integer>();

   //arraylist to hold the latetime jobs
   public static ArrayList<Integer> lateTime = new ArrayList<Integer>();
```

```
public static int capitalT = 0;
public static int smallT = 0;

public static int f1 = 0;
public static int f2 = 0;
public static int f3 = 0;

public static int capitalF[] = new int[n];

public static void main(String arg[])
{
   //find the processing time
   for(int i=0; i<n; i++)
   {
      num = Math.random();
      num1 = num * 10.0;
      p[i] = (int)num1;

      num = Math.random();
      num1 = num * 10.0;
      d[i] = (int)num1;

      num = Math.random();
      num1 = num * 10.0;
      w[i] = (int)num1;
   }
   //sort the due times
```

```
Arrays.sort(d);
System.out.print("              ");
for(int i=0; i<n; i++)
   System.out.print("Job "+(i+1)+"    ");


System.out.println();
System.out.print("processing time");
for(int i=0; i<n; i++)
{
   System.out.print(p[i]+"       ");
}
System.out.println();
System.out.print("due time      ");
for(int i=0; i<n; i++)
   System.out.print(d[i]+"       ");


System.out.println();
System.out.print("weights       ");
for(int i=0; i<n; i++)
   System.out.print(w[i]+"       ");


   for(int i=0; i<n; i++)
   {
      if(p[i] > d [i])
      {
```

```
            tardyJob += 1;
        }
    }

System.out.println();
System.out.println("No of tardy jobs = " + tardyJob);

//implement the second algorithm

for(int i=0; i<n; i++)
{
    //find the ontime jobs
    if(p[i] <= d[i])
    {
        onTime.add(i);
        smallT = smallT + p[i];//find the total time sum of ontime jobs
    }
    //find the latetime jobss
    else
    {
        lateTime.add(i);
    }

    //find the total time of all jobs
    capitalT = capitalT + p[i];
}
```

```
for(int i=0; i<n; i++)
{
   if(d[i] >= smallT)
   {
      for(int k=0; k<onTime.size(); k++)
      {
         if(i == onTime.get(k))
         {
            f1 = f1 + f1 * (smallT - p[i]);
            //capitalF[i] = capitalT[i-1]
         }
         else
         {
            f2 = f2 + w[i];
         }

      }
   }
   else
   {
      f3 = f3 + f3 * d[i];
   }
}

String tr = "";
int t;
int count = 0;
for(int j=n-1; j>=0; j--)
```

```
    {
      if(smallT < d[j])
        t = smallT;
      else
        t = d[j];
      if(f2 == f1 + w[j])
      {
        tr = tr + "    Job " + j;
        count++;
      }

      else
        t = t - p[j];
    }
    //print tardy job
    System.out.println("Tardy jobs by second algorithm");
    System.out.println("Number of tardy jobs = "+ count);
    System.out.println(tr);
  }

}
```