Chapter 1

Introduction

The problem of scheduling arises whenever work has to be done is more than the resources available. A simple example is a kitchen: If there is a single stove, variety of dishes to be cooked is more than one, and the cook has to prepare the dishes as fast as possible, then it is a sort of scheduling problem. An experienced cook knows which item is to be cooked earlier and which later so that time is saved. But in a far more complex environment like production management, computer networks, etc., we need a sound theory, based on rigorous mathematics, to handle the problems arising due to resource scarcity. Such a theory is scheduling theory.

Scheduling theory is not a mere mathematical aesthetic, but a practical requirement posed by industrial and technological development. Scheduling theory has evolved as a branch of operations research since the 1950's. In those days, this newly evolving science had major applications in industries (e.g. [48]). Today, apart from industries, scheduling theory is a crucial tool needed in computers, operating systems and several other electronic devices.

Two basic concepts required in scheduling theory are of machines and jobs – a machine executes a job. The terms 'machine' and 'job' are very general. For example, a 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 a job means a program, if a water pump is considered as a machine, then amounts of water to be pumped can be jobs, etc. In this dissertation, scheduling problems for a single machine is studied. The books referred in [11] and [7] provide a profound introduction of the scheduling theory.

The second point is *NP*-hardness. In computational complexity, the problems that have not yet been solved efficiently using any algorithm, are called *NP*-hard problems. There are several important single machine scheduling problems which are *NP*-hard. To solve such problems, practical techniques known as heuristic algorithms are used. This study includes such heuristic algorithms in the context of *NP*-hard single machine scheduling problems. Finally, a heuristic algorithm, known as tabu search, is devised for a particular *NP*-hard single machine-scheduling problem. See [15], [49], and [8] for basic theory of algorithms and complexity

Chapter 2 deals with computational complexity theory. In this chapter, the concept of good and bad algorithms is formalized. Also, some fundamental concepts of computer science are summarized.

In Chapter 3, the formal description of scheduling theory is summarized. Apart from the classical scheduling model, this chapter also gives some glimpses of some newly emerging problems and models of scheduling. However, attention is restricted to the classical model, and the details of these new ideas are not considered.

Chapter 4 mentions some easy problems in single machine scheduling. Although these problems are classic pieces in computer science literature, and can be found in most of the textbooks, they give the idea of mathematical tools used in scheduling. Moreover, single machine problems are special cases of the general scheduling problem, thus, they need attention.

In Chapter 5, there is a survey of popular techniques for handling *NP*-hard scheduling problems. Actually, these techniques are applicable to a broader class of problems, known as discrete optimization problems, of which scheduling is a special case. So again in this chapter, the context is of single machine scheduling.

In Chapter 6, heuristic algorithms are discussed. Heuristic techniques like local search and genetic algorithms are being more and more popular in computer science. Some of these techniques and the criteria for evaluating their performance are discussed.

In Chapter 7, a heuristic algorithm, known as tabu search, is devised for the *NP*-hard single machine scheduling problem, where jobs arrive over time and preemption is not allowed (i.e., jobs can not be paused). This problem is not an unconsidered problem. There are two heuristic algorithms known as Earliest Completion Time (ECT) and Earliest Start Time (EST) for this problem (see [7]). There are many other approaches of handling such a *NP*-hard problem. But in this dissertation, tabu search is chosen because it is a simple algorithm that produces better solutions, and thus getting more and more acceptance in practical areas. Furthermore, the beauty of heuristic algorithms lie in the fact that their efficiency is usually judged by experiments, and not analytically. Tabu search is more like an engineering approach than an elegant mathematical approach [24], and it is an interesting paradigm for programming. The program code of this tabu search algorithm as well as ECT and EST heuristics are given in the Appendix.

The main results of this dissertation and prospected future study are summarized in Chapter 8.

Chapter 2

Computational Complexity

Computational Complexity is a branch of computer science and applied mathematics that deals with analysis of algorithms. It deals with nature of problems solvable by algorithms, and classifies problems into several classes according to their difficulty. *NP*-hard problems are popularly taken as a synonym for 'difficult' problems. To describe the notion of *NP*-hardness, one needs a series of definitions from the theory of computation: such definitions are given from Section 2.1 up to Section 2.5. In Section 2.6, a practical example is given to describe the hardness imposed by *NP*-hard problems in practical situations.

2.1 Turing Machines and Algorithms

For a pure theoretical perspective, a computer is modeled as a Turing Machine, which basically converts one set of strings to another. A Turing Machine comprises of a finite control, a tape, and a head that can be used for reading and writing on that tape (see [39] and [26]). The formal definition of a Turing Machine is slightly involved. This study continues discussion from another viewpoint: algorithms. Though there is no precise definition of an algorithm, it is believed that algorithms and Turing Machines are equivalent. Equivalence in this context means every problem solvable by algorithms is solvable by Turing Machines and vice-versa. This definition of equivalence does not consider efficiency.

There are still other models of computations apart from Turing Machines and algorithms. Actually, the Church-Turing thesis states that all these models are equivalent (see [39] and [26]). Here, this elementary definition of an algorithm is accepted: An algorithm is any well defined computational procedure that takes some value, or set of values as input and produces some value, or set of values as output.

2.2 Asymptotic Order of Functions

There are two qualities that make a good algorithm: less time requirement and less space requirement. These space-time requirements are highly influenced by the real machine used for computation. If a bad machine is used, even a better algorithm may appear inefficient compared to a bad algorithm in a better machine. Due to these considerations, computational complexity deals with instances whose input size is very large, so that machine difference can be neglected. To describe behavior of algorithms for large input, the concept of asymptotic order is needed. Three asymptotic orders are frequently used, viz., big-O, big-omega and big-theta. Details of these concepts can be found in books like [15] and [8].

Let $t:N\rightarrow\Re^+$ and $f:N\rightarrow\Re^+$ be two functions from the set of natural numbers to the set of non-negative real numbers. The function t(n) is said to be upper bounded by f(n) and written $t(n)\in O(f(n))$ iff there is a positive real constant c and an integer threshold n_0 such that

$$t(n) \leq cf(n) \text{ for all } n \geq n_0.$$

The function t(n) is said to be lower bounded by f(n) and written $t(n)\in\Omega(f(n))$ iff there is a positive real constant c and an integer threshold n_0 such that

$$t(n) \geq cf(n) \text{ for all } n \geq n_0.$$

The function t(n) is said to be tightly bounded by f(n) and written $t(n)\in\Theta(f(n))$ iff $t(n)\in O(f(n))$ and $t(n)\in\Omega(f(n))$.

Note that $O(f(n))$, $\Omega(f(n))$ and $\Theta(f(n))$ all represent classes of functions, this justifies the use of the set-membership symbol '$\in$' for denoting asymptotic order. But popularly, the equal-to sign '=' is used instead of '$\in$'. This popular notation is used in the following discussions and later chapters.

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 to specify 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.

2.4 Problems and Encoding

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

Now some important types of problems in computational complexity theory are mentioned. A problem whose output can either be ‘yes’ or ‘no’ is called a *decision problem.* One may use any symbol instead of ‘yes’ and ‘no’. Scheduling problems belong to the class of discrete optimization problems. For this, the following definitions of an instance of optimization problem and an optimization problem are needed: An *instance of an optimization problem* is a pair (F,c), where F is any set, the domain of feasible points, c is the cost function, a mapping $c:F\rightarrow\Re^+$. The problem is to find $f\in F$ such that for all $g\in F$, $c(f) \leq c(g)$ (some optimization problem may use $\geq$ instead of $\leq$, such problems are called maximization problems). An *optimization problem* is a set I of instances of an optimization problem (see [49]).

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

Now, the concept of a discrete optimization problem is defined: An optimization problem whose all instances have finite set of feasible solutions is known as a *discrete optimization problem*. Discrete optimization problems are also known as *combinatorial optimization problems* (for example, see [49]).

To be computed by an algorithm, problem instances should be represented in a suitable way. Generally, binary strings are used for this purpose. First consider the definition of an encoding: An *encoding* of a set S of abstract objects is a mapping e from S to the set of binary strings. Generally the set {0, 1} is used as an alphabet for binary encoding. Let e be an encoding of a set S of abstract objects. Let $s \in S$, then the *length of encoding*, denoted by |e(s)|, is the number of symbols in e(s) (see [15]).

There is a special reason for choosing binary string instead of unary or others. Actually the efficiency of an algorithm is affected by the encoding used. Time and space complexities are described in terms of length of the encoded string. So, if an input has length n in a unary encoding, then in a binary encoding, its length $= \lfloor \log_2 n \rfloor + 1$, which differs exponentially from the length in unary encoding. This shows that the encoding scheme cannot be let arbitrary. However, binary encoding has an advantage over unary. Because the length of an input string of in a higher order encoding differs from length in binary encoding by a polynomial factor only. This can be seen by changing base of logarithm while converting a string in higher order encoding to binary encoding. So, if binary encoding is used instead of higher order encodings, the complexity remains same. The only trouble is with unary encoding. In the following discussions, the length of an input instance means the length of corresponding binary encoding.

2.5 Complexity Classes

2.5.1 Classes $\boldsymbol{P}$ and $\boldsymbol{NP}$

To define the complexity classes, the concept of a verification algorithm is needed. Let X be a decision problem. A verification algorithm A for the problem X takes as an input the pair (x, q), where $x \in X$ is an instance of X, and q, known as certificate, is a binary string whose size is in O(f(|x|)) , where f is a polynomial. The algorithm returns 'yes' if q is a solution of x (see [15]).

Now complexity classes are considered. There are several complexity classes in the theory of computation. But here, only the major classes relevant to this dissertation are discussed. The class of all decision problems that can be solved in polynomial time is known as the complexity class *P*. Problem belonging to *P* are said to have 'efficient' algorithms. Any algorithm having complexity of a lower order polynomial is accepted as an efficient algorithm. On the other hand, the class of all decision problems that have verification algorithms with polynomial complexity is known as the complexity class *NP*. The notation *NP* actually refers non-deterministic polynomial-time algorithms. But nowadays, the notion of non-deterministic algorithms is rarely used in the definition of *NP* class. See [8] for a description of complexity classes from an algorithmic view.

Example 2.1 The problem of sorting n numbers can be done in O(n^2) time using the quicksort algorithm in the worst case (see [15]). Thus, all sorting problems are in *P*.

Example 2.2 A vertex cover of an undirected graph G = (V, E) is a subset V' $\subseteq$ V such that if (u, v) $\in$ E, then u $\in$ V' or v $\in$ V' or both. That is, each edge touches at least one vertex in V'. The vertex-cover problem is to find such a vertex cover of minimal cardinality. This problem is *NP* (see [15])

.2.5.2 Classes *NP*-Complete and *NP*-Hard

The definition of *NP*-complete and *NP*-hard classes requires the concept of polynomial reduction. To define polynomial reduction, the definition of a polynomial-time computable function is needed: A function f:{0, 1}*→{0, 1}* is said to be *polynomial-time computable* if there exists a polynomial-time algorithm A which, for all input x$\in${0, 1}, produces the output f(x) in polynomial time (see [15]). Now comes the definition of polynomial reduction: Let X and Y be two problems. We say that X is *polynomially reducible* to Y, and write as X $\leq_P$Y, if there exist a polynomial-time computable function f:$\{0, 1\}^* \rightarrow \{0,1\}^*$ such that for all x$\in\{0, 1\}^*$, x$\in$X if and only if f(x)$\in$Y. Such a function is called a *polynomial reduction* from X to Y (see [15]).

The notion $X \leq_P Y$ is often stated as ‘Y is as hard as X’ or ‘Y is not easier than X’, etc. Informally, this means that difficulty of solving problem X is not harder than difficulty of solving problem Y. Note that this definition of polynomial reduction applies to all types of problems. Concerning decision problems, an important fact is stated below.

Lemma 2.1 Let X and Y be two decision problems such that $X \leq_P Y$, then $Y \in P \Rightarrow X \in P$.

Proof. Let $Y \in P$**.** Since $X \leq_P Y$, there exists a polynomial-time computable function $f:\{0, 1\}^* \rightarrow \{0, 1\}^*$ such that $x \in X$ if and only if $f(x) \in Y$. To solve $x \in X$, first reduce it to $f(x)$ which takes polynomial time, say $t_1(|x|)$. Now $f(x)$ is an instance of Y, so it can be solved in polynomial time, say $t_2(|f(x)|)$. Equivalently we get solution of $x \in X$ in time $t_1(|x|) + t_2(|x|)$, which is a polynomial. Hence, $X \in P$.

□

The class *NP*-complete is the set of decision problems X such that

1. $X \in NP$

2. For all $Y \in NP$, $Y \leq_P X$ (see [15]).

In simple words, *NP*-complete are the hardest problems among the *NP* class. A question arises immediately: Is the class *NP*-complete non-empty? The answer is yes. There is a famous theorem, known as Cook’s theorem, which demonstrates the existence of a *NP*-complete problem known as satisfiability problem. The satisfiability problem, denoted as SAT, states that given a set U of Boolean variables and a collection C of clauses over U, is there a satisfying truth assignment for C? (see [49])

Theorem 2.1 (Cook’s Theorem) (see [49]) The problem SAT is *NP*-complete.

Actually, hundreds of important problems have been proved to be *NP*-complete till now (see [39], [26]). Now an intuitively obvious fact is mentioned:

Theorem 2.2 $P \subseteq NP$

Proof. Let $X \in P$, then all instances $x \in X$ can be solved in polynomial time. Then one can verify X in polynomial time in this simple way: Solve X in polynomial time and check it against the given certificate. Thus, $P \subseteq NP$.

□

One may now suspect whether $P = NP$. Intuition suggests that $P \neq NP$. But this is an open question of computer science (see [14] for the formal definition of this problem). For $P = NP$, we should have $P \subseteq NP$ and $P \supseteq NP$. The first inclusion comes from Theorem 2.2. The second inclusion is an open problem, and has stunning practical consequences. Now, an important theorem of computational complexity is mentioned:

Theorem 2.3 (see [49]) If any *NP*-complete problem is polynomial time solvable, then $P = NP$. Equivalently, if any problem in *NP* is not polynomial time solvable, then no *NP*-complete problem is polynomial-time solvable.

Proof. Let $X \in NP$. From the definition of *NP*, any problem $Y \in NP$, we have $Y \leq_P X$. Now, if $X \in P$, then from Lemma 2.1, $Y \in NP$. Thus, $P = NP$.

□

If $P = NP$, then according to above theorem, all the *NP*-complete problems can be solved in polynomial time: this would be a huge enhancement in the applicability of computers. Several problems in the field of Artificial Intelligence, optimization, and algorithms in general are *NP*-complete, if a constructive proof for $P = NP$ could be found; these problems would be solved in polynomial time. But most of the researchers believe $P \neq NP$. Cook [13] mentions the reason for this: Since 1970's when this question was posed, innumerable computer scientists, mathematicians and engineers have been trying to find a polynomial time algorithm for one of the several *NP*-complete problems, but not a single algorithms has been found.

The class *NP*-hard is the class of all problems X such that for all $Y \in NP$, $Y \leq_P X$ (see [8]). In other words, there may be a problem X which is as hard as any problem in *NP*, but one may not be able to prove its *NP*-completeness, such a problem is *NP*-hard. Note that the class *NP*-hard can contain problems other than decision problems also. An optimization problem is called *NP*-hard if its decision counterpart is *NP*-complete (see [16]).

Example 2.3 The vertex-cover problem stated in Example 2.2 is *NP*-hard (see [15]). Note that this problem is not a decision problem.

Though the definitions of the classes *NP*-complete and *NP*-hard appear similar: the main differences are, *NP*-hard problems need not be decision problems, and *NP*-hard problems need not be in *NP*. But there are several reasons to study *NP*-hardness rather than *NP*-completeness. Brassard and Bratley [8] point the following reasons:

1. There are important problems that are not decision problems. For example, scheduling problems are not decision problems, but discrete optimization problems.
2. Even in the case of decision problems, there are decision problems which are *NP*-hard but not *NP*, hence not *NP*-complete.
3. For practical purposes, *NP*-hardness is the only thing to be established, because *NP*-hard and *NP*-complete both *cannot* have efficient algorithms.

The phrase ‘cannot’ is italicized intentionally. This is elaborated by the following theorem.

Theorem 2.4 (see [8]) If $P \neq NP$, then no *NP*-hard problem can be solved in polynomial time.

Proof. If we assume $P \neq NP$, the *NP*-complete problems cannot be solved in polynomial time. And since any *NP*-hard problem is as hard as a *NP*-complete problem, it cannot be solved in polynomial time.

□

Proving *NP*-completeness will be easier if one can show that a problem already known to be *NP*-complete polynomially reduces to the current problem, and for proving

NP-hardness, this will be sufficient. So, one has to explicitly prove the existence of at least one *NP*-complete problem. Fortunately, *NP*-completeness of the satisfiabilty problem SAT has an explicit proof. There are hundreds of important *NP*-complete and *NP*-hard problems (see [49] for some examples).

2.6 Hardness Imposed by NP-Hardness

In this section, an example is given to illustrate the hardness of *NP*-hard problems. Consider the famous Traveling Salesman Problem, TSP, which states that given a complete weighted graph G(V, E) with |V| = n, find a Hamilton circuit having minimum total weight, where total weight is the sum of all edges in the circuit (see [49]).

TSP is *NP*-hard, actually, the decision version of TSP is NP-complete (see [8], [49]). It is interesting to know how hard this problem is: given n vertices, there can be (n-1)! tours/circuits. If n = 10, there are 9! = 362,880 tours to be examined. If there are 30 cities (i.e. n = 30), the number of tours is 29! which is larger than10^{30}. Even if we could examine a billion tours per second – a pace far beyond the capabilities of existing or projected computers – the required time for completing this calculation would be more than a billion human lifetimes! (see [39]). This example clearly demands the necessity of evaluating near-to-exact solutions for *NP*-hard problems, because finding actual solution is practically impossible. This example thus intensifies the need of evaluating near-to-exact solutions.

Regardless the fact that all *NP*-hard/*NP*-complete problems are computationally hard, some of them may still have tolerably efficient exact algorithms. Such algorithms, known as *pseudo-polynomial* algorithms have complexities slightly higher than polynomials (see [49] for formal definition). On the other hand, there are *strongly NP*-hard and *strongly NP*-complete problems: Under the assumption $P \neq NP$, strongly *NP*-hard and strongly *NP*-complete problems can not have pseudo-polynomial algorithms (see [49]). For example, the problem TSP is strongly *NP*-hard. Thus two *NP*-hard or *NP*-complete problems may not be computationally equivalent. Due to these reasons, scheduling problems have also been classified according to the hierarchy of complexity (see [12]).

Chapter 3

Scheduling Problem

In this chapter, the basic formulation of the scheduling theory is described. The classification of scheduling problems mentioned in this chapter follows the notation used in [11]. Since the domain of scheduling theory is very wide, only single machine scheduling problems are considered as far as possible. Sections 3.1 to 3.6 include basic concepts and terminology of scheduling theory. Section 3.7 describes the extension of basic scheduling model to encompass realistic situations.

3.1 Schedules and Their Representation

Let there be m number of machines, M_i, i = 1, …, m, which have to process n jobs, J_j, j = 1, …, 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; such an assignment is called a schedule (see [11]). A schedule is often represented as a Gantt chart. Below is an example for a single machine schedule:

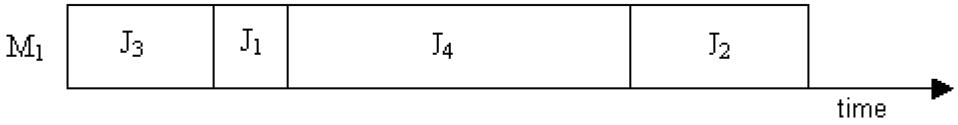


Figure 3.1 *Gantt chart for a schedule of four jobs in single machine*

In some circumstances where scheduling times of jobs are obvious or irrelevant, one can represent a schedule as a sequence of jobs. For example, the schedule shown in Figure 3.1 can be written as the sequence π = (J_3, J_1, J_4, J_2). In some cases, the machine may remain idle for a time interval; such idle intervals are depicted by simply writing ‘idle’ for that period in the Gantt chart.

There are further descriptions which specify the problem of scheduling. These descriptions are the number of machines, the types of jobs and their inter-relations, the objective function, etc. They are mentioned in the following sections.

3.2 Machine Environment

There can be a single machine, multiple machines, or in some situations, the number of machines may be unknown in advance. Let us briefly discuss the multiple machine environment, which is the general case (see [11], [7] for complete description). In the multiple machine environment, a job J_i is a set of n_i numbers of operations, O_i. It is not necessary that an arbitrary operation of an arbitrary job can be processed in an arbitrary machine: this restriction inspires to classify the multiple machine environment into two groups; parallel machines and dedicated machines.

In parallel machine model, an arbitrary operation O_{ij} of an arbitrary job J_i can be executed in an arbitrary machine M_j. Stating simply, any machine can execute any operation of any job. There are further classifications of parallel machines according to the speed of the machines, but our brief discussion will not proceed there.

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

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

Flow shop is same as open shop; only there is an additional precedence constraint: each operation O_{ij} should precede $O_{i-1,j}$ for all job J_j. Job shop is an extension of the open shop: Number of operations per job is arbitrary, and in general, the set of machines that can execute O_{ij} is different from the set of machines that can execute $O_{i,j+1}$.

3.3 Job Description

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

Another important data regarding a job J_j is its release date, r_j. Release date means the time by which the job is ready for processing. Similarly, for each job J_j, there may be weight w_j, due date d_j, and a deadline $d\square_j$. Weight of a job means its priority. Due date and deadline have this interpretation: If a job does not complete before its due date, the quality of the output detoriates. In the other hand, if a job does not complete before its deadline, the output is invalid. These information may or may not be given.

There can be precedence relation among the jobs. In general, a precedence relation is a directed acyclic graph (DAG), G(V,E), where the vertices V are the jobs, and $(J_i,J_j) \in E$ if and only if J_i precedes J_j. In specific situations, precedence relation can occur as a tree, or some other special type of DAG like sp-graphs (see [11]). A tree can be outtree or an intree. In outtree, the DAG is a rooted tree with indegree for each vertex at most one. Similarly, an intree is a rooted DAG with outdegree for each vertex at most one.

3.4 Objective Functions in Scheduling

For each job J_j, r_j is the release time, d_j is the due –date, and w_j is its weight, or priority. Given a schedule of jobs, the following parameters can be calculated (see [7] and [11]):

Completion time C_j

Flow time $F_j = C_j\text{-}r_j$

Lateness $L_j = C_j\text{-}d_j$

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

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

Accordingly, one can define objective functions, such as,

Schedule length (makespan) $C_{max} = \max\{C_j\}$

Weighted completion time $\sum_j w_j C_j$

Total completion time $\sum_j C_j$

Mean flow time $F_{mean} = (1/n) \sum_j F_j$

Flow time variance $F_{var} = (1/n) \sum_j (F_j - F_{mean})^2$

In scheduling problems, one of these objective functions has to be minimized.

3.5 The Three-Field Notation: ~ | | ¥

For specifying scheduling problems, three-field notation is popularly used. This notation is due to Graham et al. [21]. In this scheme, a problem is denoted as $\alpha \mid \beta \mid \gamma$, where the α-field describes the machine environment, β-field describes the jobs and their interrelations, and γ-field denotes the objective function. This notation is described as per the requirements of this document. For details, one can refer [11] and [7].

The α-field, in general, is $\alpha = \alpha_1 \bullet \alpha_2$, where $\bullet$ denotes string concatenation.

Parameter $\alpha_1 \in \{\phi, P, Q, R, O, F, J\}$characterizes the type of machine used. Specifially,

$\alpha_1 = \phi$: single machine (ϕ denotes empty string)

$\alpha_1 = P$: identical machines

$\alpha_1 = Q$: uniform machines

$\alpha_1 = R$: unrelated machines

$\alpha_1 = O$: dedicated machines, open shop system

$\alpha_1 = F$: dedicated machines, flow shop system

$\alpha_1 = J$: dedicated machines, job shop system

Parameter $\alpha_2 \in \{\phi, k\}$ denotes the number of machines

$\alpha_2 = \phi$: number of machines is assumed to be variable

$\alpha_2 = k$: k number of machines

The β-field, in general, is $\beta = \beta_1 \bullet \beta_2 \bullet \beta_3 \bullet \beta_4 \bullet \beta_5 \bullet \beta_6$.

Parameter $\beta_1 \in \{\phi, pmtn\}$ denotes whether preemption is allowed or not.

$\beta_1 = \phi$: preemption is not allowed

β_1 = pmtn : preemption is allowed, i.e., a job being proceesed can be paused arbitrarily, and start some another available job.

Parameter $\beta_2 \in \{\phi$, res$\}$ indicates additional resource constraints.

$\beta_2 = \phi$: no resource constraints

β_2 = res : resource constraints are given

Parameter $\beta_3 \in \{\phi$, prec, tree$\}$ denotes precedence constraints.

$\beta_3 = \phi$: no precedence constraints

β_3 = prec : precedence is given in the form of an arbitrary DAG.

β_3 = tree : precedence is given in the form of a tree

β_3 = intree : precedence is given as an intree

β_3 = outtree : precedence is given as an outtree

Parameter $\beta_4 \in \{\phi, r_j\}$ describes release dates.

$\beta_4 = \phi$: release date is zero for all jobs (or equal release dates)

$\beta_4 = r_j$: release date is given for each job

Parameter $\beta_5 \in \{\phi, p_j=p\}$ describes the processing times.

$\beta_5 = \phi$: jobs have arbitrary processing times

$\beta_5 = (p_j=p)$: all jobs processing times equal to p

Parameter $\beta_6 \in \{\phi$, d□$\}$ denotes whether deadlines are given or not.

$\beta_6 = \phi$: no deadlines

β_6 = đ : jobs have deadlines

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

3.6 Polynomial Reduction between Scheduling Problems

Many scheduling problems polynomially reduce to other problems by making changes in the α, β, and γ fields. This concept of reduction is important for obtaining complexity of scheduling problems from some older problem whose complexity is well established. The possibility of polynomial reduction is depicted in the so-called reduction graph. There are reduction graphs for all the α, β, and γ fields. However, for single machine problems, α field is always 1, so only the reduction graphs for β and γ fields are presented. Reduction graphs for all classes of scheduling problems can be found in [12].

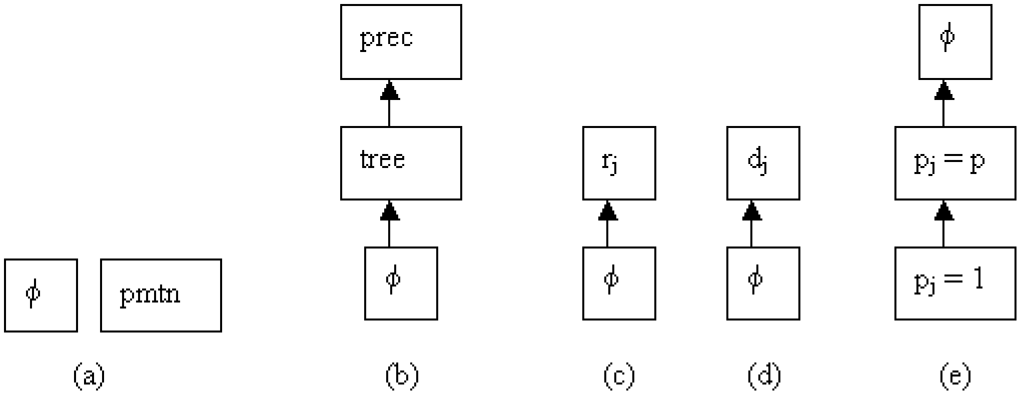


Figure 3.2 *Reduction graphs for the field [(a) Possibility of preemption, (b)Precedence constraints, (c)Release dates, (d)Due dates, (e) Processing times]*

Reduction graphs can be interpreted in this way: An arc A→B in a reduction graph means there is a polynomial reduction from A to B only if the size of B is polynomially bounded above by the size of A. For example, one can conclude that the problem $1 \mid tree \mid \sum U_j$ polynomially reduces to $1 \mid prec \mid U_j$ because there is an arc tree→prec in the reduction graph of Figure 3.2(b). Note that polynomial reduction is transitive, for example, according to graph of Figure 3.2(e), the problem $1 \mid p_j = 1 \mid L_{max}$ polynomially reduces to problem $1 \mid \mid L_{max}$.

Below is the reduction graph for the γ field. To take an example, from the graph one can infer that the problem 1 | r_j | $\sum C_j$ polynomially reduces to the problem 1 | r_j | $\sum w_jC_j$, 1 | | C_{max} reduces to 1 | | $\sum w_jU_j$, etc.

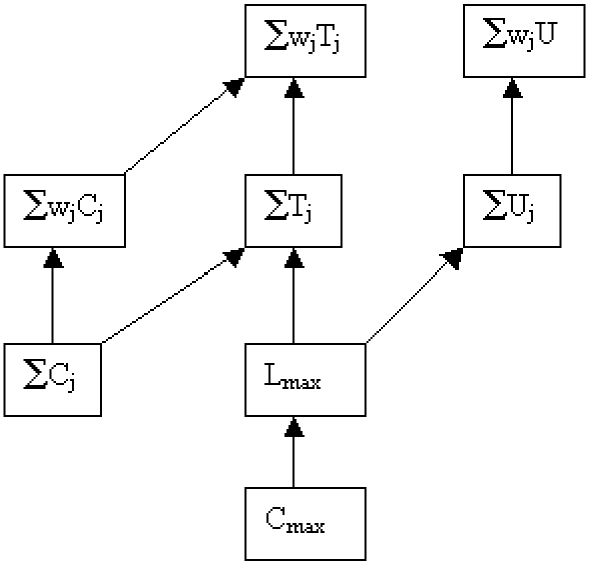


Figure 3.3 *Reduction graph for the ¥ field.*

3.7 Realistic Scheduling Problems

The scheduling model formulated in this chapter is elementary and lacks many realistic features like unavailability of job information until its arrival, resource constraints, etc There are many extensions of the basic scheduling problem, some of them are mentioned in the following sub-sections.

3.7.1 Online Problems

The discussion so far is considered with offline version of scheduling. In an online version, one does not know processing time and other relevant information of a job until it actually arrives in the system. This online model is realistic. For example, consider the online version of the problem 1 | r_j | $\sum C_j$. This problem has a deep significance in operating system design. One of the goals that the operating system seeks to achieve is to minimize the average response time, i.e., $(1/n)\sum(s_j-r_j)$, where s_j is the starting time of a job J_j (see [45]). Since $s_j = C_j$-p_j, and p_j is constant for all job j, this problem is equivalent to minimizing total completion time. Not only scheduling, many computational problems in general have online versions.

In the field of scheduling, as well as many related topics, the online versions are getting more and more attention. Many of the classical problems recur in modern online applications, but with slightly modified objective functions. Among these new objectives functions, flow time and stretch are important ones (see [29]). Flow time was defined in Section 3.3. Stretch of a job is its flow time divided by processing time. Many applications like handling requests at web servers, scheduling jobs in operating systems, parallel computing, etc. pose scheduling problems in which flow time and/or stretch has to be minimized. Jawor [29] describes the latest research on online algorithms. Unless specifically mentioned, the discussion throughout this document considers offline version of scheduling problems.

An interesting feature of online algorithms is that despite their complexity, commonly used algorithms for solving them are very simple and easy to implement. In an online environment, scheduling decision has to be taken in a very short interval of time. And such decisions have to be taken again and again. So, one does not have the time to find exact solution, simple algorithms giving reasonable solutions are acceptable. A few examples of online problems are given in Section 5.2.2.

3.7.2 Just-In-Time and Real-Time Scheduling

Just-in-time scheduling models assume the existence of job due dates and discourage early as well as tardy jobs (see [17], [18], and [40]). Take an example of a manufacturing company. If items are produced earlier than the expected date, then they have to be preserved, and thus add unwanted storage costs. On the other hand, late production can lead to fines, express delivery charges, lost sales, etc. A just-in-time schedule minimizes sum of earliness and tardiness penalties.

A special type of just in time scheduling is due date scheduling. Basically, there are two versions of this problem. In the first version, a common due date is given for all jobs, one has to find schedule minimizing lateness and tardiness penalties with respect to this due date. The second version is reverse of the first one, here, one has to determine a common due date such that the penalties are minimized. The symbols ‘d’ and ‘d_{opt}’ are added in the β field of the three field notation to indicate first and second versions due date scheduling, respectively. Unlike other scheduling problems, usually due date scheduling considers positional weights. This means, weight w_j does not correspond to job J_j, but to any job that occurs in position j of the schedule (see [17], [18], and [40]). An example of due date scheduling is presented in Section 5.31.

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 [56] provides an introduction for real-time scheduling problems in Operating Systems.

3.7.3 Set-up Times and Resource Constraints

There is a special type of scheduling problem in which jobs are classified in F families. For f = 1, …, F, each family contains N_f jobs labeled as (1, f), …., (N_f, f). All jobs are available at time zero. For j = 1, …, N_f, a job (j, f) has processing time p_{jf} and a due date d_{if}. If two jobs from the same family are scheduled contiguously, then no set-up time is necessary between these two jobs. On the other hand, a non-negative set-up time s_f is required before the processing of a job from family f if it is the first job in schedule, or if it is scheduled immediately after a job from different family. The machine can process at most one job at a time. The goal is to find a schedule minimizing some objective function, for example, maximum lateness (see [23]). This scheduling problem is an instance of a more general problem, where there can be multiple machines, and the setup time depends on many parameters (for example, see [11], [7], and [23]).

Complication of scheduling further increases when resource constraints are added. In many circumstances, in addition to the machines, additional resources may be required by the jobs. However in practice, the number of these resources is much lesser than the number of jobs. For example, in a multiprogramming operating system, many processes share resources like memory, input/output devices etc. The machine environment and job interrelation model discussed in Section 3.2 is not sufficient for analyzing scheduling problems under resource constraints.

3.7.4 Scheduling Problems in Operating Systems

Theoretically, scheduling problems posed by Operating Systems (OSs) are not different from the problems discussed till now. To be precise, scheduling problems of operating systems are just the online version of various scheduling problems. However, due to the vast scope of computer systems, it becomes necessary to mention some points. For details, books of Milenkovic [45] and Tanenbaum [56] are referred.

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 large variety. There can be multiple processors, preemption may or may not be allowed, and in almost all situations, the scheduling problems are resource constrained. Due to this variation, OS designers take an engineering approach. They select algorithms in the basis of simulation experiments. The second point to be mentioned is objective function. There is a crucial difference between manufacturing companies and computer systems at this point. A manufacturing company aims to reduce production cost, whereas an OS aims to provide a fair service to all user processes. This leads to objective functions like:

1. *Processor utilization:* This is the average fraction 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 the number of processes by schedule length.

3. *Average turnaround time:* Turnaround time is the time that elapses from the moment a program is released until it is completed by the system. If a process J_j has release time r_j and completion time C_j, then its turnaround time $\tau_j = C_j - r_j$.
4. *Average waiting time:* Waiting time is the time that a process spends waiting for the processor or some other resources. A process J_j with processing time p_j and turnaround time τ_j has waiting time $W_j = \tau_j – p_j$.
5. *Average response time:* Response time is the time taken by a process to 'response' after it is released. For a process J_j with release time r_j and start time s_j, response time is $R_j = r_j-s_j$.

Analysis of scheduling algorithms in computer system is mostly based in the elements of queuing theory. The basic queuing model is depicted below.

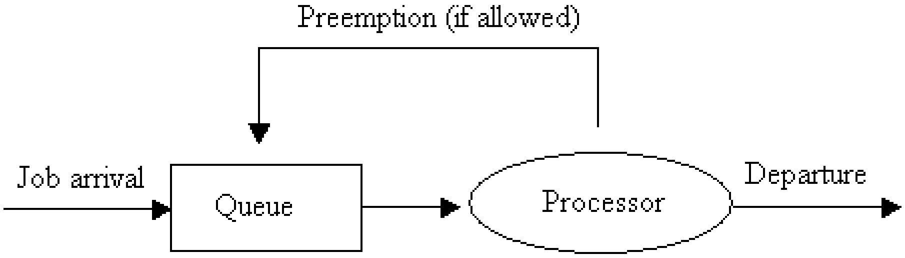


Figure 3.4 *The basic queuing model*

Jobs arrive and wait in a queue as shown above. For an OS, the queue is the main memory. Every scheduling algorithm of an OS follows this model, below are basic algorithms used is OS for uniprocessor computers.

1. *First Come First Serve (FCFS):* At any instant when machine is idle, select the available job having least release date.
2. *Shortest Processing Time (SPT): A*t any instant when the machine is idle, select the available job having least processing time. This rule is also known as the Shortest Job First (SJF) rule.

3. *Shortest Remaining Time Next (SRTN):* At each release time or finish time of a job, schedule an unfinished job which is available and has the smallest remaining processing time. SRTN rule is applicable in preemptive systems.
4. *Round Robin:* Store available jobs in a queue sorted according to release dates. Allow unit processing time to each job in the queue in the sorted order. If a new job arrives, append it to the queue. If a job completes, remove it from the queue. Round-Robin rule is applicable in preemptive system.

There are some assumptions of the queuing theory. For example, infinite numbers of jobs are assumed, and the arrival pattern of these jobs is assumed to be Poisson’s distribution. Based on these assumptions, performance of the above algorithms is evaluated (see Milenkovic [45] for details of these concepts).

This study does not include many examples of these realistic and complex problems because of two reasons. First, this dissertation deals with near-to-exact evaluation of *NP*-hard scheduling problems. Fortunately, as far as the offline versions are considered, the techniques discussed in chapters 6 and 7 are applicable to these complex problems also (see for example [40], [23] and [50]). The second reason is, a proper treatment of these problems require sophisticated mathematical tools beyond the scope of this short study.

Chapter 4

Polynomially Solvable Single Machine Scheduling Problems

As discussed in Chapter 2, problems that can be solved in time bounded above by a small degree polynomial are known as easy problems. There are many easy single machine scheduling problems having practical as well as theoretical importance. The algorithms and theorems regarding these problems have become classic pieces in the scheduling literature, and thus, often available in textbooks. However, some of these problems are presented here, because the algorithms and theorems regarding these problems give a glimpse of the vast field of scheduling.

4.1 1 | prec | f_{max}

For this problem, associated with each job j is a monotone non-decreasing cost function f_j. Each f_j is evaluated in unit time for any value of the argument. The objective function is the maximum cost, $f_{max} = \max\{f_j(C_j)\}$. The algorithm for this problem is due to Lawler [32].

Let N = {1, ….., n} be the set of all jobs and let $S \subseteq N$ be the set of unscheduled jobs. Let $p(S) = \sum_{j \in S} p_j$, i.e., total processing time of all unscheduled jobs. The scheduling rule is: 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 the schedule.

Now Lawler's algorithm for 1 | prec| f_{max} is stated. The precedence constraints is given by the adjacency matrix $A = (a_{ij})$ where $a_{ij} = 1$ if and only if j is a direct successor of i. By n(i), the number of immediate successors of i is denoted.. The optimal schedule is denoted as π : $\pi(1), \ldots.., \pi(n)$ where $\pi(i)$ denotes the job in position i.

Algorithm [32] Lawler's algorithm for $1 \mid prec \mid f_{max}$

```
Begin
    1. For i :=1 to n do $n(i) := \sum_{j=1}^{n} a_{ij}$
    2. $S := \{1,\ldots, n\}$, $p := \sum_{j=1}^{n} p_j$
    3. For k = n down to 1 do
  Begin
    4. Find job $j \in S$ with $n(j) = 0$ and minimal $f_j(p)$ value
    5. $S := S \setminus \{j\}$
    6. $n(j) := \infty$
    7. $\pi(k) := j$
    8. $p := p - p_j$
    9. For i := 1 to n do
           If $a_{ij} = 1$ then $n(i) := n(i)-1$
End of algorithm
```

The complexity of this algorithm is $O(n^2)$. Now the proof for the correctness of this algorithm is presented.

Theorem 4.1 [32] Lawler's algorithm for $1 \mid prec \mid f_{max}$ constructs an optimal sequence.

Proof. Enumerate the jobs in such a way that 1, 2,…., n is the sequence constructed by the algorithm. Let $\sigma : \sigma(1),\ldots,\sigma(n)$ be an optimal sequence with $\sigma(i) = i$ for $i = n, n-1,.., r$ and $\sigma(r-1) \neq r-1$ where r is minimal. The sequence σ is in the following situation:

σ :	......	r-1	k	.....	j	r	....	n

It is possible to schedule r-1 immediately before r. Therefore, r-1 and j have no successor in the set {1,…., r-1}. This implies $f_{r-1}(p) \leq f_j(p)$ with $p = \sum_{i=1}^{r-1} p_i$ because 1,…., n was constructed by the algorithm. Thus, the schedule obtained by shifting the block of jobs between r-1 and r an amount of p_{r-1} units to the left and processing r-1 immediately before r is again optimal. This contradicts the minimality of r.

□

Example 4.1 Consider 5 jobs $J_{1, \ldots\ldots,} J_5$ with the following processing times.

j	1	2	3	4	5
p_j	2	4	6	3	5

The precedence relation is given below in Figure 4.1(a) and the monotone functions f_i's are given in Figure 4.1(b).

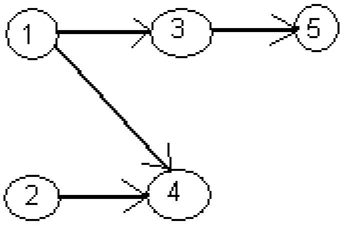


Figure 4.1 (a) *The precedence relation in Example 4.1*

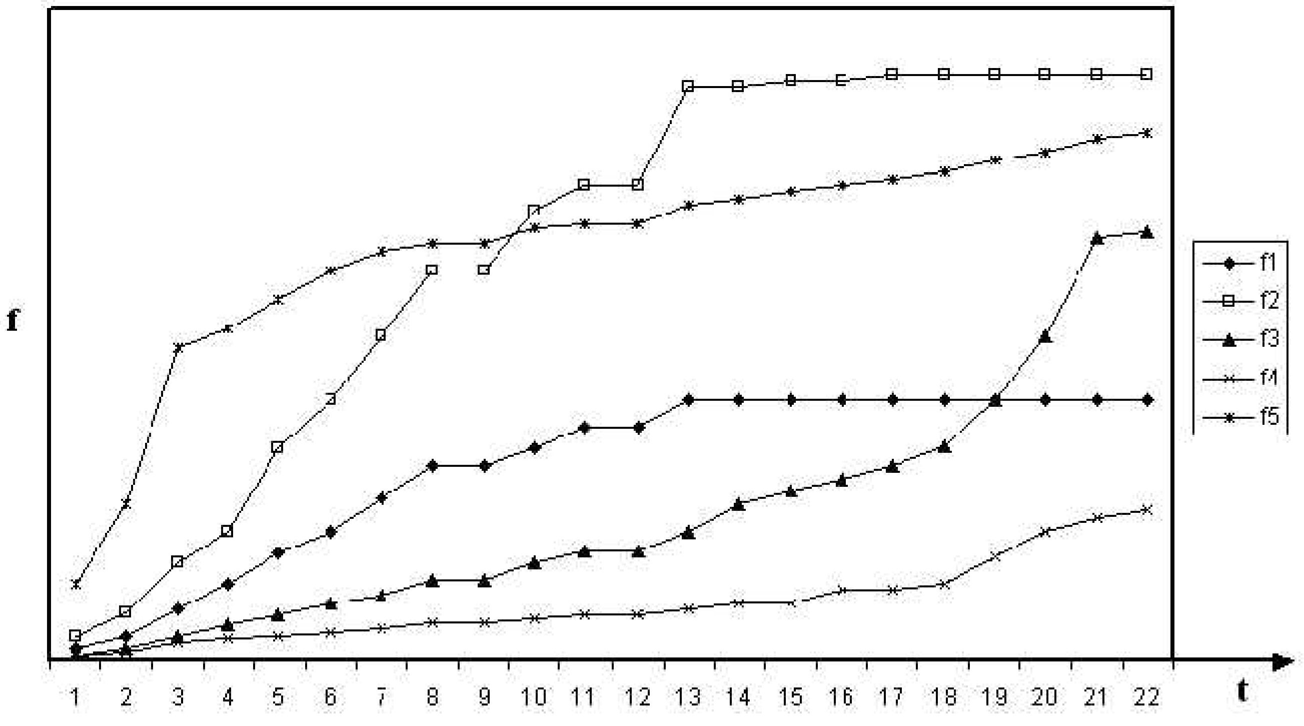


Figure 4.1 (b) *The monotone functions f_i's for Example 4.1*

Initially, t = total completion time = $p_1 + p_2 + p_3 + p_4 + p_5 = 20$.

At t = 20, J_5 and J_4 have no successors, but $f_4(20)$ is minimal (see above figure), so J_4 is chosen, the partial solution denoted by $\pi = (J_4)$, and $t := 20 - p_4 = 17$.

At $t = 17$, J_2 and J_5 have no successors, but $f_5(17)$ is minimal, so J_5 is chosen, $\pi = (J_5, J_4)$, and $t := 17\text{-}p_5 = 12$. At $t = 12$, J_2 and J_3 have no successors, but $f_3(12)$ is minimal, so J_3 is chosen, and $\pi = (J_3, J_5, J_4)$, and $t := 12\text{-}p_3 = 6$.

At $t = 6$, J_3 and J_1 have no successors, but $f_1(6)$ is minimal, so J_1 is chosen, $\pi = (J_1, J_3, J_5, J_4)$, and $t := 6\text{-}p_1 = 2$.

At $t = 2$, only J_2 is available, so final schedule is $\pi = (J_2, J_1, J_3, J_5, J_4)$.

4.2 1 | | ġw_jC_j

This problem can be solved using the Weighted Shortest Processing Time (SWPT) rule. The SWPT rule is to sort jobs in non-decreasing order of p_j/w_j. This SWPT rule produce an optimal solution for the problem 1 | | $\sum w_jC_j$. The optimality of SWPT rule can be proved as a consequence of a more general theorem due to Lawler [36]. 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 π of jobs, find a permutation π^* such that

$$f(\pi^*) = \min \{ f(\pi) \mid \pi \text{ is a permutation of } N \}.$$

In some special cases one can find a transitive and a complete relation Y on the set of jobs N such that for any two jobs $J_i, J_k \in N$, and for any permutation of the form $\alpha J_iJ_k\delta$,

$$J_i \, Y \, J_k \;\Rightarrow\; f(\alpha J_iJ_k\delta) \, Y \, f(\alpha J_kJ_i\delta). \qquad (4.2.1)$$

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

Theorem 4.2 [36] If f admits the task interchange relation Y, then an optimal permutation π^* can be found by ordering the task according to Y.

Proof. If f admits a task interchange relation Y, then (4.2.1) holds. This means, whenever J_k and J_i are adjacent in a sequence, $J_k \, Y \, J_i$, and J_k comes before J_i, then J_k and J_i can be interchanged without increasing the total cost. Repeating this for all possible pairs, one gets an optimal schedule which is ordered according to Υ.

□

The optimality of SWPT rule can be proved as a consequence of theorem 4.2.1:

Theorem 4.3 (see [7]) SWPT rule produces optimal sequence for the problem $1 \mid\mid \sum w_jC_j$.
Proof. Let us define a relation Υ on the set of jobs such that the SWPT rule produces a sequence which is ordered according to Υ. More specifically, define Υ such that for any two jobs J_i and J_k

$$J_i \Upsilon J_k \quad \Leftrightarrow \quad p_i/w_i \leq p_j/w_j \qquad (4.2.2)$$

Now, if one can show that Υ is a task-interchange relation for the objective function $\sum w_jC_j$, then according to Theorem 4.2.1, SWPT rule produces an optimal sequence. For two jobs J_i and J_k, let $J_i \Upsilon J_k$ and consider a sequence $\alpha J_iJ_k\delta$. If the last task in the subsequence α finishes at time t, cost of $\alpha J_iJ_k\delta$ is

$$cost_1 = w_i(t + p_i) + w_k(t + p_i + p_k) + b$$

where b considers all the costs of tasks in subsequences α and δ. If J_i and J_k are interchanged, the resulting sequence is $\alpha J_kJ_i\delta$, whose cost is

$$cost_2 = w_k(t+p_k) + w_i(t + p_k + p_i) + b.$$

From (4.2.2), $cost_1 \leq cost_2$. Thus, optimality is proved.

□

Example 4.2 This example demonstrates the SWPT rule. Consider five jobs $J_1,\ldots,J_5$ with following information:

j	1	2	3	4	5
p_j	8	9	12	5	14
w_j	4	3	3	5	2
p_j/w_j	2	3	4	1	7

Sorting according to the p_j/w_j values the optimal solution is $\pi = (J_4, J_1, J_2, J_3, J_5)$. The total cost $\sum w_jC_j = 5.(5) + 4.(5+8) + 3.(5+8+9) + 3.(5+8+9+12) + 2.(5+8+9+12+14) = 341$.

4.3 1 | p_j = 1 | ģw_jU_j

The objective function in this problem involves unit penalty U_j, this means, for each job J_j due date d_j is given. Here an algorithm for 1 | p_j = 1 | $\sum w_jU_j$ 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 maximal. 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 form S.

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

```
Algorithm (see [11])  1 | $p_j$ = 1 | $\sum w_jU_j$
Begin
    t = 1, S = $\phi$
    for i = 1 to n do
        if  $d_i \geq t$  then
            add i to S,  t = t + 1
    if there exists a job $k \in S$ with $w_k < w_i$ then
    begin
        Delete job k from S where k is the largest index such that $w_k$ is
        minimal
        add i to S
    end
End of algorithm
```

If the scheduled jobs in S is organized as a priority queue with respect to the w_j value, the complexity of this algorithm is O(n logn).

Theorem 4.4 (see [11]) Algorithm 1 | p_j = 1 | $\sum w_jU_j$ provides an optimal schedule.

Proof. Let S be the sequence of jobs scheduled early by the algorithm ordered according to their indices. Let S* be the corresponding sequence for an optimal schedule coinciding with S as long as possible. Let k be the first job in S* which does not belong to S. 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 (i included) at the time k was eliminated. Due to step 6 of the algorithm $w_j > w_k$ for all job $j \in J$. Thus all $j \in J$ 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 4.3 To demonstrate how the algorithm for $1 \mid p_j=1 \mid \sum w_jU_j$ constructs the set of early jobs consider five jobs $J_1,\ldots,J_5$ with following information:

j	1	2	3	4	5
w_j	3	5	1	2	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.

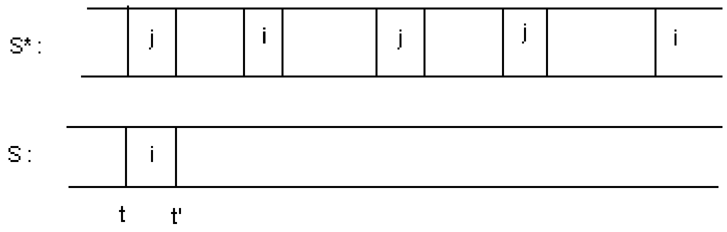
The jobs in S are to be scheduled as per the sequence $\pi = (J_3, J_5, J_1)$. J_2 and J_4 can be scheduled afterwards in any order.

4.4 1 | r_j, pmtn | ġC_j

This problem can be solved in $O(n^2)$ time using the Shortest Remaining Time Next (SRTN) rule, which can be stated as: At each release time or finish time of a job, schedule an unfinished job which is available and has the smallest remaining processing time. This is known as Smith’s rule.

Theorem 4.5 [5] The SRTN rule constructs optimal schedule for the problem $1 \mid r_j, pmtn \mid \sum C_j$.

Proof. Let S be the schedule constructed by applying the SRTN rule and let S* an optimal schedule. Assume that the both schedules coincide until time t. Then the following situation occurs:



Let the job i in S which starts at time t be processed up to time t'. Let j be the job in S* which starts at time t. According to SRTN rule, the remaining processing time of j is not smaller than that of i. Further, between t and t' there is no release time. Now in S* all intervals of both jobs i and j which do not start before time t are eliminated. After this these parts are rescheduled in the empty time slots starting at time t by first scheduling the remaining parts of i and then the remaining parts of j. The schedule created in this way is still optimal. This interchange process is repeated until S and the new optimal schedule coincide up to time t'.

□

Example 4.4 This example demonstrates the SRTN rule. Consider five jobs J_1,….,J_5 with the following information.

j	1	2	3	4	5
p_j	6	2	4	2	1
r_j	2	0	1	4	2

Let A denote the set of available jobs.

At t = 0, J_2 releases, A = {J_2}, J_2 is scheduled.

At t = 1, J_3 releases, A = {J_2, J_3}, but J_2 has shortest remaining time (SRT), J_2 scheduled.

At t = 2, J_1 and J_5 release, A ={J_3, J_1, J_5}, J_5 has SRT so it is scheduled.

At = 3, J_5 completes, A = {J_3, J1}, J_1 has SRT so it is scheduled.

At t = 4, J_4 releases, A = {J_3, J_1, J_4}, J_4 has SRT so it is scheduled.

At t = 6, J_4 completes, A = {J_3, J_1}, J_3 has SRT so it is scheduled.

At t = 10, J_3 completes. J_1 is the only job remaining, so it is scheduled. The final schedule is depicted in the following Gantt chart.

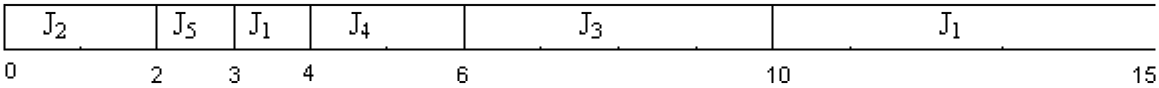


Figure 4.2 *Solution of Example 4.4*

4.5 1 | outtree | ġw_jC_j

In this problem, the precedence relation is given as an outtree. The algorithm for this problem is due to Adolphson and Hu [1]. For each job j, let $q_j = w_j/p_j$ and let S(j) be the set of successors (not necessarily immediate) of j including j. For a set of jobs I $\subseteq$ {1, 2, …., n} define

$$w(I) = \sum_{i \in I} w_i$$

$$p(I) = \sum_{i \in I} p_i$$

$$q(I) = w(I)/p(I)$$

Let i→j mean job i precedes job j in the given outtree. The following theorem gives the main idea for constructing algorithm for 1 | outtree | $\sum w_jC_j$.

Theorem 4.6 [1] Let i, j be jobs with i→j and $q_j = \max \{q_k \mid k \in S(i)\}$. Then there exists an optimal schedule for 1 | outtree | $\sum w_jC_j$ in which i is processed immediately before j.

Besides the given outtree for precedence, the algorithm uses an outtree in which each node represents a sequence or a subsequence of jobs in {1, 2, …, n}. Initially, this outtree is a shadow of the given outtree, i.e., each node represents a sequence of single job corresponding to that node of the given outtree. Then the algorithm goes on merging the nodes in this way: Each node i represents a sequence π_i and the corresponding set of jobs A_i, where i is the first job in π_i. In the general step, the algorithm selects a node j different from the root with maximal q(j) value. Let f be the unique father of j in the original outtree. Then the algorithm finds a node i of the current tree with $f \in A_i$. Then the nodes i and j are merged, replacing π_i and A_i by $\pi_i \bullet \pi_j$ (where $\bullet$ represents string concatenation) and $A_i \cup A_j$, respectively. Then the children of node j become the children of node i. The algorithm is given below, where E(i) denotes the last job of π_i, P(i) denotes the predecessor of i, and the root of the outtree is assumed to be i = 1. Further, $w_1 = -\infty$ is taken so that root is not selected while searching for maximal q-value.

```
Algorithm [1]    1 | outtree | $\sum w_j C_j$
Begin
        1.  w(1) = $-\infty$
        2.  for i := 1 to n do
                    E(i) := i
                    $A_i$ := i
                    q(i) := w(i)/p(i)
        3.  L := {1, …, n}
        4.  while L $\neq$ 1 do
            begin
        5.  find $j \in L$ with largest q(j) value
        6.  f := P(j)
        7.  find i such that $f \in A_i$
        8.  w(i) := w(i) + w(j)
        9.  p(i) := p(i) + p(j)
        10. q(i) := w(i)/p(i)
        11. P(j) := E(i)
        12. E(i) := E(j)
        13. $A_i$ := $A_i \cup A_j$
        14. remove j from L
            end
End of algorithm
```

This algorithm can be implemented in O(n logn) time if a priority queue is used for the q(j) values and an efficient union-find algorithm for sets is used in Steps 7 and 13. The following theorem states the optimality of the algorithm.

Theorem 4.7 [1] Algorithm 1 | outtree | $\sum w_jC_j$ calculates optimal sequence.

Proof. The proof is done by induction on the number of jobs. The algorithm is clearly correct for a single job. Let o be the problem with n jobs. Let i and j be the first jobs merged by the algorithm. Let o' be the problem that results after merging i and j, i.e., i is replaced by I = (i, j) with w(I) = w(i) + w(j) and p(I) = p(i) + p(j). Let *R* be the sequences of the form

$$\pi : \pi(1), ..., \pi(k), i, j, \pi(k+3), \ldots, \pi(n)$$

and let *R'* be the sequences of the form

$$\pi : \pi(1), ..., \pi(k), I, \pi(k+3), \ldots, \pi(n).$$

According to theorem 4.6, *R* contains an optimal schedule. Now, let $f_n(\pi)$ and $f_{n-1}(\pi')$ denote the objective values $\sum w_jC_j$ for π and π', respectively. Then,

$$\begin{aligned} & f_n(\pi) – f_{n-1}(\pi') \\ &= w(i)p(i) + w(j)(p(i) + p(j)) – (w(i) + w(j))(p(i) + p(j)) \\ &= - w(i)p(j) \end{aligned}$$

This means $\pi \in R$ is optimal if and only if the corresponding sequence $\pi' \in R'$ is optimal. But π' has only n-1 jobs. Thus, by the induction hypothesis, the sequence constructed by the algorithm most be optimal.

□

Example 4.5 This example illustrate algorithm 1 | outtree | $\sum w_jC_j$. Consider an instance of 6 jobs with precedence relation and other parameters given in the following figure.

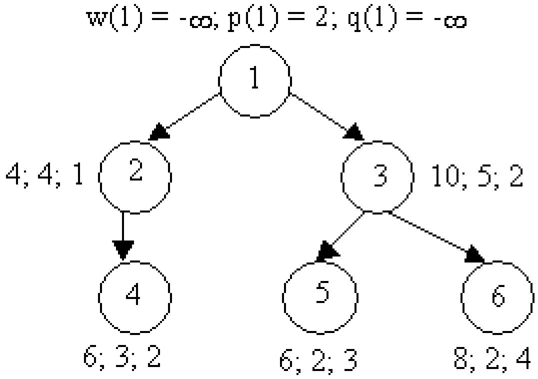


Figure 4.3(a) *Outtree for Example 4.5*

Node 6 has the maximum q-value of 4, so it is merged with node 3 resulting in the following tree.

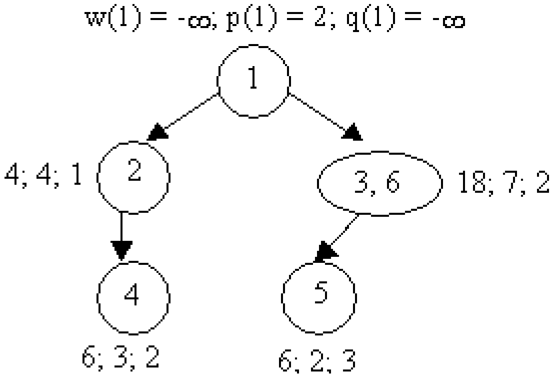


Figure 4.3(b) *Tree after merging nodes 3 and 6*

Continuing in this way, one obtains the following single noded tree, which represents optimal solution.

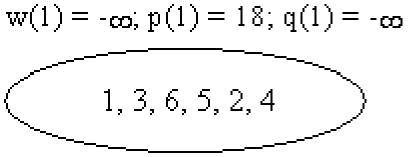


Figure 4.3(c) *Final tree representing the solution*

This algorithm can be slightly modified to solve the problem 1 | intree | $\sum w_jC_j$ [1]. A 1 | intree | $\sum w_jC_j$ – problem P can be reduced to a 1 | outtree | $\sum w_j'C_j'$ - problem *P'* with

- i is a successor of j in *P'* if and only if j is a successor of i in *P*
- $w_j' = -w_j$ for j = 1, …, n

Then a sequence π: 1, …., n is feasible for *P* if and only if π':n, …., 1 is feasible for *P'*. Further, it can be proved that a sequence π is optimal for *P* if and only if the reverse sequence π' is optimal for *P'*.

4.6 Some More Problems

There are several other easy problems of importance in single machine scheduling. Description of all of them is not possible here. A brief summary of some more polynomially solvable problems covered during this study is given below.

$1 \mid \mid L_{max}$

Jackson [28] proved that this problem can be solved in O(nlogn) time using the Earliest Due Date (EDD) rule. The EDD rule is simply to schedule jobs in non-decreasing order of due dates. The optimality of this rule has a simple proof: Let π be an optimal schedule and π^* be an EDD schedule. If $\pi \neq \pi^*$ then there exist two jobs J_j and J_k with $d_k \leq d_j$, such that J_j immediately precedes J_k in π, but J_k precedes J_j in π^*. Since $d_k \leq d_j$, interchanging the positions of J_j and J_k in π can not increase the value of L_{max}. A finite number of such changes transforms π into π^*.

$1 \mid r_j, p_j = 1 \mid L_{max}$

Horn [27] proved that this problem can be solved in O(nlogn) time using the EDD rule.

$1 \mid prec, p_j = 1 \mid L_{max}$

This problem can be solved using the algorithm due to Monma [46]. The basic idea of the algorithm is this: The first step is to modify due dates. If a job J_i precedes J_j and $d_i' := d_j – p_j < d_i$, then replace d_i by the modified due date d_i'. The next step is to make the due dates non-negative. If all due dates are already non-negative, then nothing is to be done. Otherwise the minimum modified due date is subtracted from all modified due dates. This second modification assures that $L_{max} \geq 0$. After the modification of due dates in this manner, the remaining part of the algorithm is developed using the following ideas:

1. The jobs are processed in the interval [0, n]. This implies that no job J_j with $d_j \geq n$ is late even if it is processed as the last job. Because $L_{max} \geq 0$, these jobs have no influence on the L_{max} value.

2. Sort the jobs in non-decreasing order of due dates using bucket sort. Construct the buckets B_k as

$$B_k = \begin{cases} \{ j \mid d_j = k \} & \text{if } 0 \le k \le n\text{-}1 \\ \\ \{ j \mid d_j \ge n \} & \text{if } k = n \end{cases}$$

Modification of the due dates can be done in O(n+e) time, where e is the number of edges in the precedence graph [46]. Bucket sort can be done in O(n) time (see [15]). Thus the overall complexity is O(n+e).

1 | | ģU_j

This problem can be solved in O(nlogn) time using the algorithm due to Moore [47]. The algorithm for 1 | | $\sum U_j$ is very much similar to the algorithm described in Secion 4.3 for the problem 1 | p_j = 1 | $\sum w_jU_j$. Moore’s algorithm [47] for 1 | | $\sum U_j$ constructs a maximal set S of jobs which complete on time. The optimal solution then consists of the jobs in S scheduled according the EDD rule, followed by the late jobs in any order. The set S is constructed by this rule [47]: Add the jobs in S in order of non-decreasing due dates. If the addition of job a 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 form S.

1 | | ģw_jU_j with Agreeable Weights

The problem 1 | | $\sum w_jU_j$ is NP-hard, this was proved by Karp [30]. However, there is a special case of this problem where the weights are agreeable. Weights are called agreeable if for two jobs J_i and J_j, $p_i < p_j$ implies $w_i \ge w_j$. Lawler [33] devised a O(nlogn) time algorithm for this special case. The idea is again similar to that described in Section 4.3 and the previous problem (1 | | U_j), i.e., construct an optimal set of S jobs on time, schedule the jobs of S in EDD order and late jobs in any order. Refer [33] for details.

1 | r_j | ģU_j

The algorithm for this problem also constructs a maximal set S of jobs completed on time, which are scheduled in EDD order, and late jobs are scheduled in arbitrary order.

Lawler [35] devised an algorithm based on this idea and proved that it gives optimal solution for $1 \mid r_j \mid U_j$. Refer [35] for the details.

$1 \mid pmtn, prec, r_j \mid f_{max}$

The objective function f_{max} for this problem is defined in Section 4.1. This problem can be solved in $O(n^2)$ time using the algorithm of Baker et al. [6]. The algorithm follows three major steps:

1. If a job J_j is a successor of job J_i and $r_i + p_i > r_j$, then job J_j can not start before $r_j' = r_i + p_i$. So, replace r_j by r_j'. In this way, all release dates are modified.
2. Schedule the jobs in non-decreasing order of modified release dates. This decomposes the jobs into blocks, where a block is a minimal set of jobs processed without idle time in between them.
3. Find optimal solution for each block separately. The resulting set of blocks will be the optimal schedule.

For Step 3, Baker et al. [6] use a recursive procedure.

From the problems described in this chapter, it can be seen that many of these polynomially solvable problems have somewhat similar algorithms. And secondly, it should be noted that research works on these polynomial solvable single machine problem are still going on. The website of Brucker and Knust [12] maintains a list of classic as well as latest scheduling problems, their status and corresponding references.

Chapter 5

Handling *NP*-Hard Scheduling Problems

In this chapter a summary of general approach of tackling NP-hard problems taking examples of single machine scheduling problems is given.

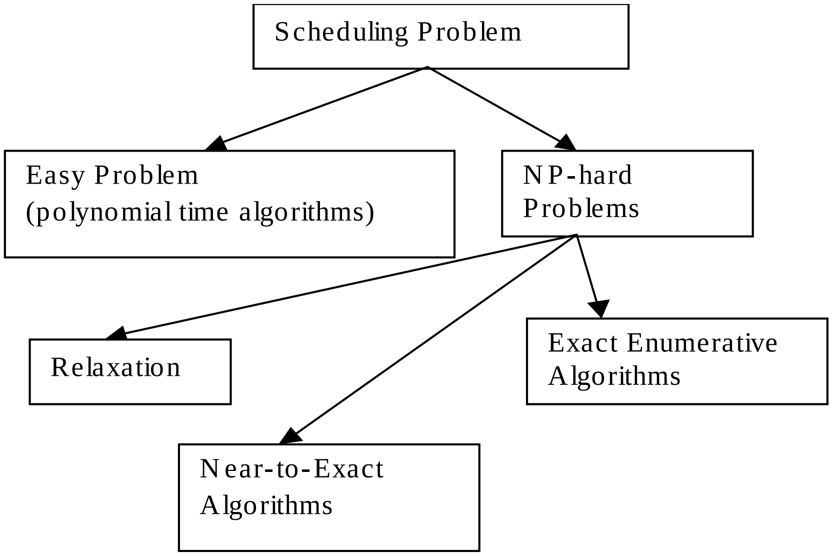


Figure 5.1 *Schematic view of scheduling complexity*

Above figure summarizes the complexities of scheduling problems and approaches for solving them. In the figure, easy problems are the problems of class *P*. Few polynomially solvable single machine problems were described in the previous chapter. In this chapter, the techniques of tackling *NP*-hard problems are briefly mentioned. Note that the techniques of relaxation, near-to-exact algorithms and exact enumerative algorithms are generic: they are applicable to discrete optimization problems in general. However, the discussion is again in the context of single machine scheduling.

As discussed in Chapter 2, hardness of all *NP*-hard problems is not same. Knowing the level of NP-hardness of a problem, one can select an appropriate algorithm. Though this is not a rule, usually exact algorithms based on dynamic programming (Section 5.3.1) are applied to pseudo-polynomially solvable problems. For example, the problem $1| | \sum w_jU_j$ has been proved to be *NP*-hard and pseudo-polynomially solvable by Karp [30]. Sahni [53] developed an exact pseudo-polynomial algorithm for $1| | \sum w_jU_j$ based on dynamic programming. On the other hand, for a strongly *NP*-hard problem, one seldom tries to compute exact solution. Rather, one applies approximate/heuristic algorithms (Section 5.2 and Chapter 7). But practical situations can demand exact solutions of even these strongly *NP*-hard problems. In such situations, branch-and-bound algorithms (Section 5.3.2) are used. Branch-and-bound techniques can give exact solutions for small instances of these strongly *NP*-hard problems. For example, the problem $1 \mid \mid \sum w_jT_j$ is strongly *NP*-hard [37]. Babu et al. [4] designed a branch-and-bound algorithm for this problem; this algorithm can find solutions for problem instances with number of jobs up to 50.

5.1 Relaxation

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

(i) Allowing preemptions: e.g. the problem $1 \mid r_j \mid \sum C_j$ is *NP*-hard [37], but its preemptive version, $1 \mid r_j; \text{pmtn} \mid \sum C_j$ can be solved in O(nlogn) time, n being the number of jobs [5].

(ii) Assuming unit processing time: e.g. the problem $1 \mid \mid \sum w_jU_j$ is *NP*-hard [37], but the problem $1 \mid p_j=1 \mid \sum w_jU_j$ can be solved in O(nlogn) time (see [11]).

(iii) Assuming equal release dates: e.g. the problem $1 \mid r_j \mid L_{max}$ is *NP*-hard [37], but if $r_j = r$ for all jobs J_j, then it can be solved in $O(n^2)$ time (see [11]).

(iv) Assuming certain precedence relation: e.g. the problem $1 \mid prec \mid \sum w_jC_j$ is *NP*-hard for a general DAG [34], but the problem $1 \mid sp \mid \sum w_jC_j$ can be solved in O(nlogn) time (see [7]).

5.2 Near-To-Exact Algorithms

As discussed in Chapter 2, till now there are no efficient algorithms for *NP*-hard problems. Even for instances of moderate size, one has to obtain solutions that are not exact, but practically tolerable. Basically, there are two classes of algorithms for obtaining near-to-exact scheduling (see [7] and [8]):

- Approximation algorithms: These algorithms provide a theoretical guarantee for the quality of the obtained solution.
- Heuristic algorithms: No such theoretical guarantee can be given. The quality of solutions is determined by simulation experiments and actual implementation.

Heuristic algorithms are considered in Chapter 6. In this section, approximation algorithms for both offline and online problems are mentioned.

The performance of approximation is measured by approximation ratio, which is, in general, a function of the size of the input instance. Let A be an algorithm. 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., $\max(c/c^*, c^*/c) \leq \rho(n)$ (see [15]). Regarding online version of scheduling as well as other optimization problems, the concept of competitive ratio is introduced. Let A_O be an online algorithm. For any instance of size n, let c_A be the cost of solution obtained by A_O and c be the cost of optimal solution for the corresponding offline problem. Then A_O is said to have a *competitive ratio* of $\rho_c(n)$ if $c_A \leq \rho_c(n)c$ (see [3]).

5.2.1 Approximation Algorithms for Offline 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 $1 \mid r_j \mid \sum F_j$. Kellerer et al. [31] have 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; here is a summary:

1. Convert the problem $1 \mid r_j \mid \sum F_j$ to $1 \mid r_j; pmnt \mid \sum F_j$ by allowing preemptions.

2. Solve 1 | r_j; pmtn | $\sum F_j$ using the Shortest Remaining Processing Time rule (see [11]).
3. From the solution of the 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.

It is not always that design of approximation algorithms for scheduling is always ad hoc. Savelsbergh et al. [54] make an empirical analysis of several approximation algorithms based on linear programming formulation for the problem 1 | r_j | $\sum w_jC_j$. They conclude that these techniques usually have complexity of O(nlogn), n being the number jobs, and have very reasonable approximation ratio.

5.2.2 Approximation Algorithms for Online Problems

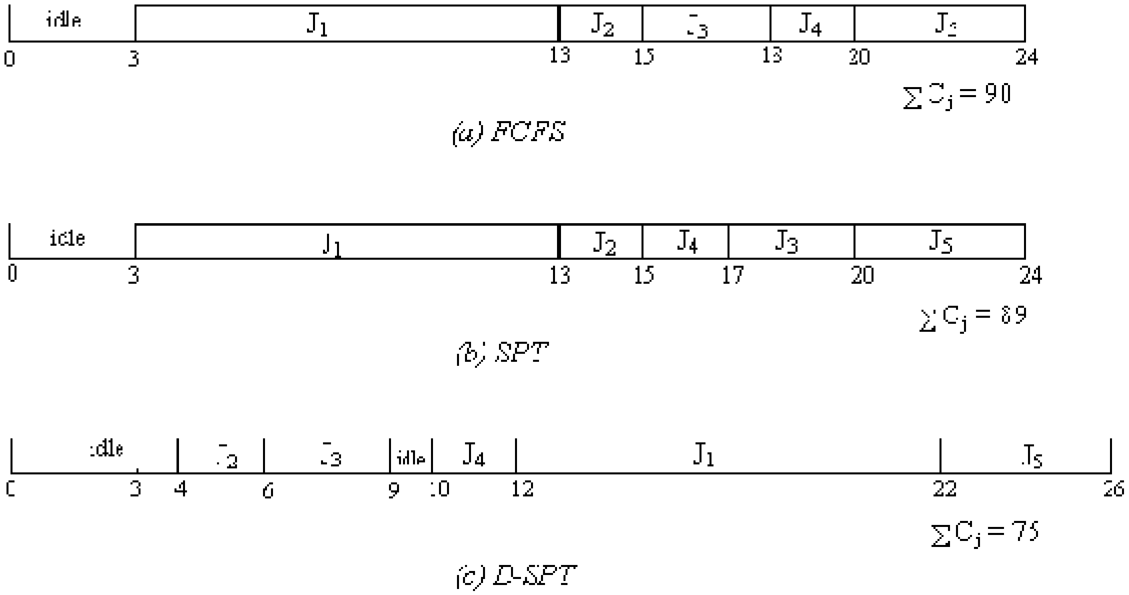
Online algorithms are getting larger attention by researchers of the scheduling theory. Consider the problem 1 | r_j | $\sum C_j$, even the offline version of this problem is *NP*-hard [37]. Regarding its online version, some popular approaches for solving it are the FCFS (First Come First Serve) and SPT (Shortest Processing Time) rules (see Section 3.7.4). Mao et al. [42] proved that 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, this is a very pessimistic result. Again, for the same problem Hoogeven and Vestjens [25] proved that the D-SPT (Delayed Shortest Processing time) rule gives a 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: [25] If the machine is idle and a job is available at time t, determine an unscheduled job with smallest processing requirement, say J_i. If there is a choice, take the job with smallest release date. If $p_i \leq$ t, then schedule J_i, otherwise wait until time p_i, or a new job arrives, whichever happens first.

Example 5.1 This example illustrates FCFS, SPT and D-SPT rules. Consider 5 jobs J_1, .., J_5 with following information.

j	1	2	3	4	5
r_j	3	4	5	10	14
p_j	10	2	3	2	4

The following figure shows the schedules obtained by these three rules. Note that FCFS and SPT rules schedule job J_1 immediately after it is released at time 3, but D-SPT keeps waits till the arrival of J_2, because processing requirement of $J_1 = 10$, which is too large in this context.



(a) FCFS

(b) SPT

(c) D-SPT

Figure 5.2 *Schedules and total completion times given by FCFS, SPT and D-SPT rules in Example5.1*

In the same paper, Hoogeven and Vestjens [25] further proved that there can be no online algorithm for the problem 1 | r_j | $\sum C_j$ having competitive ratio less than 2. Their result was generalized by Anderson and Potts [3], who proved that the D-SWPT (Delayed Shortest Weighted Processing) rule has a competitive ratio of 2 for the more general problem $1 \mid r_j \mid \sum w_j C_j$. D-SWPT is very much similar to D-SPT, the difference is due to the weights:

D-SWPT rule: [3] Suppose that the machine is available at time t. We chose from among the available job as a job J_j with the lowest value of the ratio 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.

Anderson and Potts [3] further proved that there can be no algorithm for the online version of the problem $1 \mid r_j \mid \sum w_j C_j$ having competitive ratio less than 2.

5.3 Exact Enumerative Algorithms

These algorithms try to find the exact solution. By enumeration, one understands an exhaustive visit of all possible solutions. Thus, for *NP*-hard problems, these methods are applicable only for small instances.

If we model the solution space as a graph, then the common graph search algorithms, DFS (Depth First Search) and BFS (Breadth First Search) are examples of enumerative algorithms. They are often called blind search, for they do not use the knowledge of the problem domain. Whenever we use extra knowledge of the problem domain for guiding an enumerative search, then such an enumeration is kwon as an implicit enumeration. Dynamic programming and branch-and-bound algorithms are popular techniques that use implicit enumeration. Several scheduling algorithms have been implemented using this approach.

5.3.1 Dynamic Programming

This is a modification of the divide-and-conquer approach. But unlike the divide-and-conquer technique, which is top-down in nature, dynamic programming is bottom-up.

Further, dynamic programming intelligently enumerates all visited sub-problems to avoid repeated computations. To be precise, dynamic programming has the following general steps (for example, see [15]):

1. Characterize the structure of an optimal solution,
2. Recursively define the value of an optimal solution,
3. Compute the value of an optimal solution in a bottom-up fashion,
4. Construct an optimal solution from computed information.

The first three steps give the cost of the optimal solution. The fourth step is optional, it can be omitted if one needs the value only, not the optimal solution. The following example illustrates how optimal solutions can be characterized and recursively defined in the context of a scheduling problem.

Example 5.2 *Dynamic programming applied to problem 1 | d | w_E ģE_j + w_T ģT_j*

In the problem 1 | d | $w_E \sum E_j + w_T \sum T_j$, d is the common due-date of all jobs. w_E and w_T are penalty associated with early and tardy jobs, respectively. In the following discussion, given a number x, the symbol x^+ denotes max{0, x}.

Assume that there are n jobs enumerated such that $p_1 \le p_2 \le \ldots\ldots \le p_n$.

Let $\sigma(k, e)$ = optimal sequence for jobs $J_1, J_2, \ldots., J_k$

$Z(k, e)$ = corresponding objective value.

Now, optimal solution can be recursively defined in the following way (see [11]).

<u>*Base condition:*</u>

$$Z(1, e) = \begin{cases} w_E\,(e-p_1) & \text{if } p_1 \le e \le d \text{ (if early)} \\ w_T(p_1 - e) & \text{if } e < p_1 \text{ (if late)} \end{cases}$$

$$\sigma(1, e) = 1$$

<u>*Recursive definition:*</u>

Given Z(k-1, e) and σ(k-1, e), add job J_k in the following way:

<u>*Case 1*</u>: J_k scheduled at the beginning

Cost of scheduling k at front = $w_E E_k + w_T T_k = w_E(e-p_k)^+ + w_T(p_k-e)$

Cost of scheduling remaining jobs = $Z(k-1, e-p_k)$

So, cost of scheduling J_k at first is $Z_1(k, e) = w_E(e-p_k)^+ + w_T(p_k-e) + Z(k-1, e-p_k)$

Correspondingly, $\sigma(k, e) = J_k \bullet \sigma(k-1, e)$, where $\bullet$ denotes string concatenation.

Case 2: J_k scheduled at last

Cost of first k-1 jobs = Z(k-1, e)

So, cost of scheduling J_k at last, $Z_2(k,e) = w_E E_k + w_T T_k + Z(k-1, e)$

$$= w_E(e-C_k)^+ + w_T(C_k-e) + Z(k-1, e)$$

where $C_k = p_1 + p_2 + \ldots. + p_k$, the completion time of k.

Correspondingly, $\sigma(k, e) = \sigma(k-1, e) \bullet J_k$

Thus in general, Z(k, e) and σ(k, e) can be recursively defined as follows: For k =2 to n and $0 \le e \le d$,

$$Z(k, e) = \min\{Z_1(k, e), Z_2(k, e)\}$$

and

$$\sigma(k, e) = \begin{cases} J_k \bullet \sigma(k-1, e) & \text{if } Z_1(k, e) \le Z_2(k, e) \\ \sigma(k-1, e) \bullet J_k & \text{otherwise} \end{cases}$$

The algorithm requires the Z and σ values for negative e also, the complete discussion can be found in [11].

5.3.2 Branch-and-Bound Algorithms

Branch-and-bound is a technique of exploring an implicit graph, which is most often a tree. Such graphs are called as state-space graphs. Usually each node in the search tree for a branch-and-bound algorithm represents a set of feasible solutions. The parent node represents the set of all feasible solutions. As the tree expands, the cardinality of the sets represented by the nodes go on decreasing, and finally, the leaves become singleton sets representing probable solutions, among them is the optimal solution. The task is to find a path from the root to a leaf having optimal solution as efficiently as possible (see [7], [8], [11] and [49]).

However, there is no standard of representing the solution space in a search tree in branch-and-bound paradigm. In many situations, a search node may represent a single solution, or even a partial solution. Even the path from the root to a leaf may represent a feasible solution in some implementation. Similarly, there is no standard of search technique also, though breadth first search is popular. The main essence of branch-and-bound strategy lies in the pruning of the search tree to minimize steps for searching the solution. For this, two procedures are essential, branching and bounding (see [7], [8], [11], [49]):

Branching: Given a node, this procedure generates its children. Branching procedure conceptually divides the solution space into sub-spaces, usually mutually exclusive.

Bounding: Branching procedure is also called as elimination criteria. If a node represents a set of feasible solution, the bounding procedure calculates the lower bound for cost of solution in that set: if this lower bound is greater than the best cost found so far, this node need not be expanded.

Branch-and-bound greatly reduces the search complexity, but still goes exponential in the worst case (see [49]). Efficiency of this technique depends upon the lower bounding sub-algorithm. If the lower bound of the solutions in a current node obtained from this sub-algorithm is near to the highest lower bound, then the search may be quickly guided to the optimum solution. On the other hand, evaluating such a lower bound may be time-consuming, hence decrease in overall efficiency. In general: it is next to impossible to give any idea of how well the technique will perform on a given problem using a given bound (see [8]). The following example illustrates the concepts of search tree and branching/bounding schemes in the context of a scheduling problem.

Example 5.3 *Branch-and-bound applied to problem* $1 \mid r_j, d\square_j \mid C_{max}$

The *NP*-hard problem $1 \mid r_j, d\square_j \mid C_{max}$ can be solved using the branch-and-bound algorithm due to Bratley et al. [9]. All possible task schedules are implicitly enumerated in this way: The root is assumed to be on level 0. Given n jobs, there are n children of the root. This first level represents the jobs that can occur at the first position of the schedule. Similarly, there can be n-1 children of each first level node. This second level represents the jobs that can occur at the second position of the schedule, and so on. In this way, the search tree is completed. A path from the root to a leaf represents a schedule, but this schedule may not be feasible because some of the jobs in the path can miss the deadline. Consider three jobs J_1, J_2, and J_3 with the following information.

J	1	2	3
p_j	3	2	4
r_j	2	5	1
$đ_j$	10	7	6

The following figure is the search tree for this problem instance generated as described above. The numerical value in each node indicates the completion time of the corresponding job. In the first level, completion time of each job is sum of its release and processing times. In other levels, the completion time of a job is sum of its processing time and the completion time of its parents.

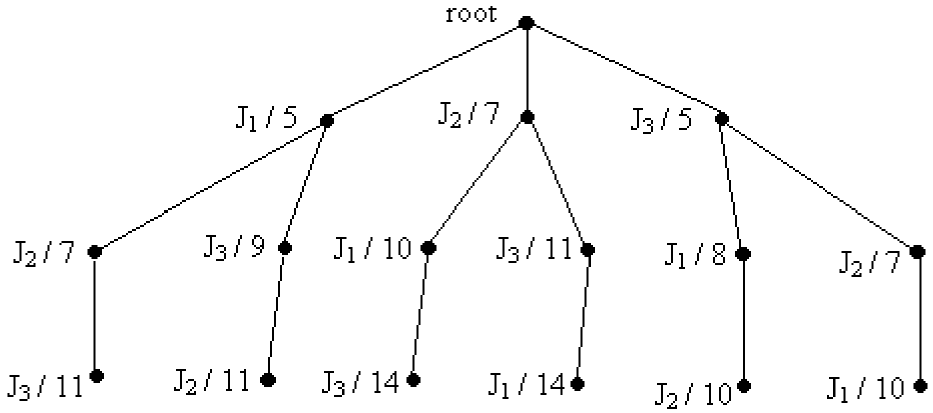


Figure 5.3 *Search tree for Example 5.3*

From the above search tree, (J_3, J_1, J_2) and (J_3, J_2, J_1) both have minimal C_{max} value. But in the former schedule, J_2 misses its deadline and hence invalid. The latter schedule is the optimal one. This example demonstrates representation of the search tree, and branching schemes for $1 \mid r_j, đ \mid C_{max}$. The bounding procedure of Bratley et al. [9] is based on the following ideas:

1. If the completion time associated with a node exceeds its deadline, the sub-tree belonging to that node is excluded from further examination.

2. If the completion time C_j of a job J_j at level k is less than or equal to the smallest release date r_{min} among yet unscheduled jobs, then there is no need to enter another branch of the tree, i.e., there is no need to backtrack beyond level k. This is because the best schedule for the remaining n-k jobs can not be started earlier than r_{min}, and hence not earlier than C_j.

This is just an outline of techniques for tackling *NP*-hard scheduling problems. One can refer the references in above discussions for further details.

Chapter 6

Heuristic Algorithms

Heuristic algorithms are frequently used for finding near-to-exact solution of *NP*-hard problems. This chapter includes heuristic algorithms popular in optimization literarature. Though these algorithms do not provide any theoretical guarantee for the quality of obtained solution, they are frequently used chiefly because of their generality. Heuristic techniques are applicable to almost any kind of optimization problems. A heuristic algorithm, also simply called a heuristic, is often devised from experience. From empirical analysis, one may observe that certain algorithm is good in most of the cases, such an algorithm is called a heuristic. According to the book of Blazewicz et al. [7], a good heuristic should have the following two qualities:

1. Complexity of the algorithm should be bounded above by a small degree polynomial.
2. The solution should be *close enough* to optimal solution. This quality of solution is evaluated empirically, considering the worst or mean case behavior.

Heuristic algorithms can be deterministic or probabilistic (see [8]). Given a problem instance, a deterministic algorithm will produce the same solution on every execution. On the other hand, given a problem instance, a probabilistic algorithm may produce different solution in different executions. Probabilistic algorithms use pseudo-random number generators for generating probability distributions. As illustrated in the following sections, most of the heuristic algorithms use some sort of probabilistic techniques. However, the distinction of deterministic and probabilistic heuristics is not of importance for our purpose.

6.1 Local and Global Optima

Analysis of heuristic algorithms needs the concepts of neighborhood, local and global optima. These concepts are elaborated in books like [49]. First of all, the definition of a neighborhood is needed. Given an optimization problem with instances (F, c), where F is the set of feasible solutions and c the cost function, a *neighborhood* is a mapping $N:F\rightarrow 2^F$. The definition can be interpreted in this way: Search for the optimal solution begins with an initial feasible solution. At any step, given a $s \in F$, one has to consider solutions 'near to' s in the solution space. This proximity is given by N(s). Local optimum is defined using the concept of a neighborhood. Given an instance (F, c) of an optimization problem and a neighborhood N, a feasible solution $f \in F$ is called locally optimal with respect to N if $c(f) \leq c(g)$ for all $g \in N(f)$.

The exact solution of any optimization problem is called as the global optimum. In the definition of global optimum, the term 'global' reminds that the solution is best among all locally optimum value. But note that the concept of global optimum does not demand the definition of any neighborhood.

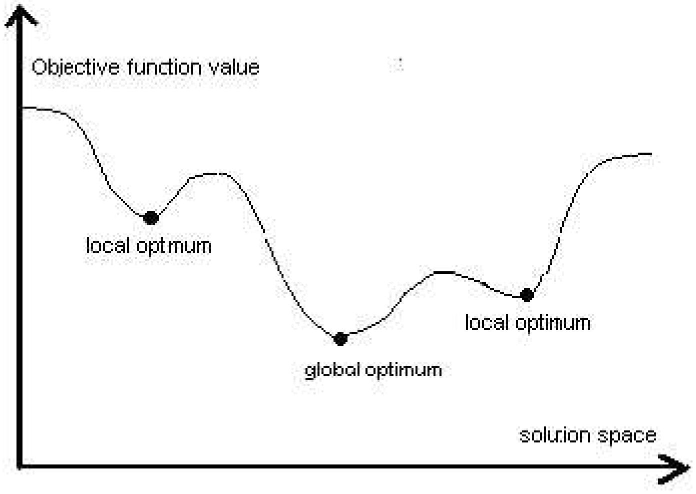


Figure 6.1 *Concept of local and global optimum*

Above figure demonstrates the concept of local and global optimum in context of minimization problems. The proximity of an optimum is often called as a valley, this terminology is clarified by the figure (for maximization problems, valleys are replace as hills). Based on these definitions, the following types of heuristic algorithms are seen in literature:

1. Algorithms that evaluate near-to-exact value for the global optimum.
2. Algorithms that evaluate the exact value for a local algorithm
3. Algorithms that merge the technique of above two types of algorithms

Perhaps due to inexpensiveness of memory and speed of modern computers, the third approach is much popular nowadays. Actually, in the current literature, no such distinctions are made

6.2 Genetic Algorithms

Genetic algorithms fall on the first category of the classification mentioned in the previous section, i.e., these algorithms search for near-to-exact value of the global optimum solution. However, as we shall soon see, the basic genetic algorithm does not require the definition of a neighborhood, and hence at least conceptually, genetic algorithm has no relevance to locality. In this section, the basic genetic algorithm is described taking example of the problem 1 | | F_{var}. The genetic algorithm for this problem has been implemented by Gupta et al. [22], their heuristic is taken for the example.

Genetic algorithms have been developed as an analogy of evolution in nature. Thus they require the concepts of a chromosome, a fitness function, crossover, mutation, population and selection. The two operations, crossover and mutation are analogues of reproduction. Fitness function is used to simulate the process of natural selection.

Chromosome A chromosome is an encoding of a feasible solution. Classically, this encoding is a binary encoding. But binary encoding is not suitable for all types of problems. So, in general, feasible solutions are encoded using a finite alphabet. For the problems 1 || F_{var}, a chromosome is taken as a sequence of integers in {1, .., n}, where n is the total number of jobs. Each integer represents a job, and the position of that job in the sequence represents the corresponding position of the job in the schedule.

Fitness function A fitness function gives the quality of a feasible solution. Generally fitness is defined in such a way that higher the fitness value, better the quality of a feasible solution. For the problem 1 || F_{var}, let g(x) be the actual cost for a feasible solution x, i.e., the flow time variance of the schedule represented by x. Then the fitness of x, f(x), is defined by Gupta et al. [22] as:

$$f(x) \quad = \begin{cases} C_{max} - g(x) & \text{when } g(x) < C_{max} \\ \\ 0 & \text{otherwise} \end{cases}$$

where C_{max} denotes the maximum completion time of a job in schedule x.

Crossover Crossover is a binary operation. Given two feasible solutions, f_1 and f_2, known as parents, the crossover operation produces two new feasible solutions, o_1 and o_2, know as children. This is done by interchanging portions of strings f_1 and f_2 in this way: randomly select two positions, known as crossover positions, and swap the substring between these two points to produce to children. The crossover operation implemented by Gupta et al. [22] is illustrated by this example. Consider an instance of 10 jobs for the problem 1 || F_{var}. Let f_1 be the sequence 9,8,4,5,6,7,1,3,2,10 and f_2 be 8,7,1,2,3,10,9,5,4,6. The crossover of f_1 and f_2 produces children o_1 and o_2 as described below:

f_1 : 9, 8, 4, | 5, 6, 7, | 1, 3, 2, 10 $\qquad$ f_2 : 8, 7, 1, | 2, 3, 10, | 9, 5, 4, 6

o_1 : 9, 8, 4, 2, 3, 10, 6, 5, 7 $\qquad$ o_2 : 8, 10, 1, 5, 6, 7, 9, 2, 4, 3

First, the crossover points for the parents f_1 and f_2, denoted by the symbol '|', are selected randomly. Then the sub-sequences between the crossover points are considered: For f_1 this subsequence is (5, 6, 7) and for f_2 (2, 3, 10). Now, considering the jobs in these subsequences, job 5 corresponds to job 2, 6 to 3 and 7. So the permutations (5, 2), (6, 3) and (7, 10) are applied to the parents: These permutations convert f_1 into o_1 and f_2 into $o_{2.}$

Population and generation Let P(t) be a set of n_P feasible solutions at any iteration t. In genetic terminology, P(t) is called population and t the generation.

Mutation Mutation is a unary operation. Given a feasible solution f, randomly swap two jobs in f. Only few solutions in a given population are mutated. This choice is generally random, or uses some special probability distribution function.

Selection Given a population P(t), the selection operator returns two feasible solutions for doing crossover. This choice of selection is often random, but sometimes, fitness function may also be considered. For the problem 1 || F_{var}, selection is made by taking two solutions using uniform distribution. Fitness value is not considered.

Description of a genetic algorithm can be found in texts [41] and [52].

Algorithm (see [41]) *Genetic algorithm*

```
Begin
    t := 0
    Initialize P(t)
    While some stopping condition is not reached
    Begin
        While population size n_P is not reached
        Begin
            Select two parents from P(t)
            Crossover
            Store children in P'(t)
        End of while
        Sort P'(t) according to the fitness values
        Select best n_P solutions from P'(t) and save in P(t)
        Randomly choose some solutions from P(t)
        Mutate the chosen solutions replace their original versions in P(t)
    End of while
End of algorithm
```

The solutions in the initial population are distributed uniformly in the solution space. This concept is illustrated in the following figure.

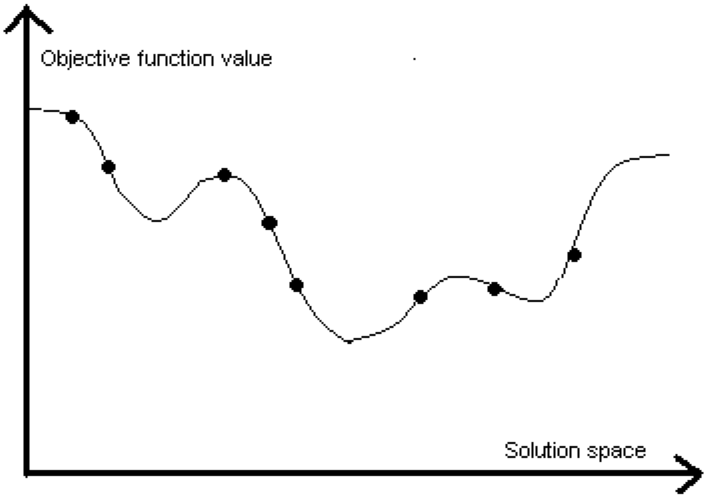


Figure 6.2 *Conceptual illustration of initial population*

In above figure, the black dots denote initial feasible solutions. After several generations, the population centers about the local minima as demonstrated by the following figure.

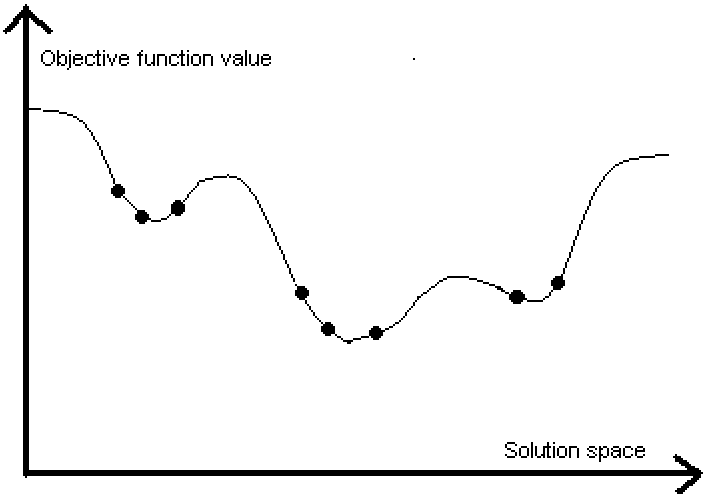


Figure 6.3 *Population after several generations*

Gupta et al. [22] implemented this genetic algorithm for 1 || F_{var} as discussed above in this section. They considered various population sizes, probability distribution, and other factors. The conclusion is, the obtained solutions were within error of less than 0.2% of the optimal solution [22].

6.3 Local Search Heuristics

As mentioned earlier, a local search algorithm finds exact value of a local optimum. The solution space is generally a directed acyclic graph. The basic local search algorithm is mentioned below. Recall that N(x) represents the neighborhood of x.

Algorithm (see [24]) *Local search*

```
Begin
    1. select a feasible solution x
    2. find x'∈N(x) such that cost of x' < cost of x
    3. if no such x' can be found, x is the local optimum and the method stops
    4. otherwise let x = x' and go to step 2.
End of algorithm
```

Here, it is implicitly assumed that the objective function has to be minimized. For maximization problems, one have to replace the ‘less than’ symbol ‘<’ in step 2 by the ‘greater than’ symbol ‘>’. Above algorithm is known as descent method. For maximization problems, the same algorithm is known as hill climbing.

A local optimum solution may differ largely form the global optimum value. So, one must try to escape trapping within a neighborhood. For this various techniques are used: If one knows that the problem has a few local optima, then restarting the search a few number of times using random initial solutions may work. But is the case of *NP*-hard problems, the number of local minima grow exponentially with the input size, thus this simple modification will not work.

There are many modifications of the basic local search algorithm. Among them, simulated annealing and tabu search are popular. These techniques basically modify step 2 of the local search algorithm stated above. They are discussed briefly in the following sub-sections.

6.3.1 Simulated Annealing

As genetic algorithms, simulation annealing is also an analogue of some natural phenomena. Simulated annealing was actually developed as a tool for some research in thermodynamics; more precisely, to simulate cooling process of a metal.

Simulated annealing modifies Step 2 of the naïve local search algorithm in the following way:

1. Instead of selecting best s' from N(s), s' is selected randomly
2. s' is accepted with the probability min{1, exp(-ΔE/T}**,** where the parameter ΔE is called as the badness of the move. It is defined as

   $$\Delta E = \text{cost of } s' - \text{cost of } s.$$

   Parameter T is known as temperature, which decreases with the iteration.

It is evident that the probability of accepting s' decreases exponentially with the badness ΔE. Since temperature T decreases with iteration, the probability also decreases with iteration. Assuming ΔE to be constant, one can see, the probability of selecting s' decreases as iteration increases. This means that bad moves are likely to be allowed at initial iterations when the temperature is high. As iteration increases and temperature decreases, best moves are likely to be chosen. Due to this, at higher level of iterations, the search gets trapped in one of the local optima. However, if T decreases slowly enough, the algorithm finds the global optimum with probability approaching unity (see [52]).

The algorithm is presented below.

```
Algorithm  (see [52]) Simulated annealing
Begin
    Select an initial solution s
    For t := 1 to infinity do
    Begin
        If T = 0 then return s
        s' := a randomly selected solution from N(s).
        $\Delta$E := cost(s') -cost(s)
        if $\Delta$E < 0 then s := s'
        else s := s' only with probability $e^{\Delta E/T}$
    end
End of algorithm
```

Simulated annealing has two major drawbacks

1. It may revisit previously examined solutions, and this repetition can occur several numbers of times, thus the search oscillates about a local optimum. Note that *NP*-hard discrete optimization problems have exponential number of feasible solutions, so, listing all visited solutions is a very inefficient and practically impossible remedy for this drawback.
2. Simulated annealing is not intelligent. Though it tries to escape local optima, this effort is not guided by the knowledge of the problem domain.

6.3.2 Tabu Search

Tabu search overcomes both drawbacks of simulated annealing listed in the previous section. The basic idea behind tabu search is that adding short-term memory to a local search improves its ability to locate optimal solutions. The word ‘tabu’ (or sometimes ‘taboo’) literally means ‘cultural prohibition for some type of activity’. In the context of Tabu Search, the word ‘tabu’ has this meaning: a recently visited solutions cannot be revisited. Let us first introduce some basic terms used in Tabu Search.

Tabu List: It is a data structure which keeps track of recently visited solutions and their quality. Generally actual solutions are not stored in the tabu list, because it may be costly in terms of space and time. Instead of the solutions, the significant steps leading towards solutions are stored. Such a step is called a move. A move already in the tabu list is said to be a tabu move. The concept of a move is elaborated below.

Move: Assume s be the current best solution. Suppose a better solution s’ is found in N(s), the neighborhood of s. Then, the key difference between s and s’ is called as a move. One can also think a move as an operation that converts s to s’. For example, in many graph problems, a commonly used move is to replace an edge in the current solution with another edge not in the solution.

Aspiration condition: Sometimes a move leading to a better solution may be a tabu move. So, in addition of checking whether a move is tabu or not, one has to further check whether that move satisfies some condition known as aspiration condition. If a tabu move satisfies the aspiration condition, it is accepted for further operations, otherwise neglected.

Tabu search was introduced by Fred Glover in late 1980’s. Since then, it has been applied in several discrete optimization problems. Till now, many variations of Tabu Search have emerged. The basic tabu search algorithm is shown below.

```
Algorithm Basic Tabu Search (see [11])
Begin
    s := an initial solution
    best_cost = cost(s)
    TabuList = empty
    while some stopping condition is not reached
    begin
        select s’ from N(s)
        Move := the move leading s to s’
        If Move is not is TabuList OR Move satisfies aspiration condition
            If cost(s’) < cost(s)
            Begin
                best_cost  := cost(s’)
                s := s’
                insert  Move in TabuList
            End
    End of while
End of algorithm
```

The basic tabu search algorithm mention above avoids unwanted oscillation about a local optimum. But it is still not intelligent. The intelligence of tabu search comes from two operations, viz., intensification and diversification, described below.

Intensification: Intensification of a search means restricting the search to some narrow region of the solution space where probably the global optimum lies. Obviously, this requires knowledge of the problem domain, hence, intensification can be termed as an intelligent operation. Although there is no standard guiding how to perform intensification, it is often done by restarting the tabu search, beginning with a solution obtained form the basic tabu search. Sometimes, size of the tabu list is also reduced while performing intensification.

Diversification: Diversification is done to escape local optimum. The simplest way to do it is to perform several random restarts. That is, for several times, randomly select an initial solution and perform the basic tabu search. However, as with intensification, there is no specific rule of implementing diversification. One has to make his own scheme.

Now, the complete tabu search algorithm is presented:

```
Algorithm (see [24]) Tabu Search
Begin
    Perform basic tabu search
    While some stopping condition is not reach
    Begin
        Intensify and perform basic tabu search
        Diversify and perform basic tabu search
    End
End of algorithm
```

6.3.3 Some Issues Related to Tabu Search

Since a partial objective of this dissertation is to design a tabu search algorithm, some issues related to tabu search are briefly mentioned below.

Levels of memory: Tabu list, introduced in Section 5.1, is a short-term memory which keeps the track of recently visited solutions. The complete tabu search algorithms performs the basic tabu search for several iterations, performing intensifications and diversifications. This suggests to use a long-term memory which keeps track of good moves found in each iteration. In general, tabu search can use several levels of memory (see [20]).

Length of the tabu list: If the length of the tabu list is kept small, then the problem of oscillation about a local optimum may still exist. On the other hand, larger sized lists decrease overall efficiency of the search. Mazure et al. [43] designed tabu search algorithm for the SAT problem and experimentally concluded that optimal length of tabu list grows linearly with the number of variables in this particular problem. In general, no standard is known. Tabu lists with length near to the input size often work well [43].

Candidate list strategies: Instead of evaluating all possible moves in a current neighborhood, the efficiency of the search can be greatly improved if good moves are evaluated before the bad moves. Such good moves are called candidate moves. Moreover, in many problems, even the size a neighborhood is large, so it is better to restrict on a small part of the neighborhood at first. A candidate list maintains good moves of a neighborhood. For details of candidate lists in scheduling refer [50].

Local search heuristics are extensively used in the field of scheduling. For example, consider the *NP*-hard scheduling problems $1 \mid \mid \sum w_j T_j$ and $1 \mid r_j \mid \sum w_j T_j$. Babu et al. [4] designed a branch-and-bound algorithm for $1 \mid \mid \sum w_j T_j$ which works up to instances of 50 jobs. However, for instances larger than 50 jobs, no branch-and-bound algorithm is available for both of these problems. But Braune et al. [10] have implemented basic local search tabu search and a genetic algorithm for these problems, and claim that their algorithms give reasonable accuracy.

6.4 Hybrid Algorithms

Genetic algorithms are general and have high range of applicability. But they are weaker in results. Local search heuristics like simulated annealing and Tabu Search are specific, they give better results for particular problems, but lack generality. Due to this, many hybrid algorithms are described in the current literature. By hybrid, algorithms that encompass the essentials of both genetic algorithms and local search heuristics are understood.

Consider the Quadratic Assignment Problem, QAP: Given two sets A = $\{a_1, a_2, .., a_n\}$ and B with $|B| = n$, and a cost function $c:A\times B\rightarrow\Re$, find a one-to-one mapping $f:A\rightarrow B$ such that

$$\sum_i c(a_i, f(a_i))$$

is minimized (see [44]). In simple words, assign each member of A to a member of B such that each $a\in A$ is assigned with a unique $b\in B$ and the total assignment cost is minimized.

QAP has many areas of application. For example, every single machine scheduling problem without preemption and without precedence constraints is an instance of QAP. To see this, let A be the set of jobs, that is A = $\{J_1, J_2, \ldots, J_n\}$, and let B = $\{1, 2, .., n\}$. Then a one-to-one mapping from A to B represents a schedule of jobs in J. The problem is to find such a schedule minimizing the total weighted penalty.

McLoughlin and Cedeno [44] have designed a hybrid algorithm for QAP by combining genetic algorithm and Tabu search. They give the name “Enhanced Evolutionary Tabu Search” for their algorithm. This algorithm obtained a reasonable solution for the QAP.

Combination of genetic algorithm and simulated annealing is also seen in the field of scheduling. Affenzeller and Mayrhofer [2] design such a hybrid algorithm. They applied their algorithm to a routing problem and obtained reasonable results.

6.5 Performance Evaluation of Heuristic Algorithms

Efficiency of heuristic algorithm is examined experimentally. This is generally done by comparing obtained results with benchmark suites.

Taillard [55] has designed benchmarks for the flow-shop, job-shop and open-shop problems. However, the described scheme of designing benchmark is applicable to all sorts of scheduling problems. The scheme used by Taillard [55] is as follows.

1. Generate job data like processing times, weights, etc., using uniform distribution. Uniform distribution is obtained by the linear congruential pseudorandom number generator.

2. Obtain near-to-exact solution using Tabu search. The easy point is, benchmark is designed only once, so the algorithm need not be fast, but give as better solution as possible.

It is not always necessary to use uniform distribution for generating job data. Depending upon the application domain, other distributions may be used. For example, in operating systems, it is better to use Poisson's distribution for generating release dates of processes (see [45]).

Though the methods of evaluating the efficiency of heuristic algorithms depend on computer simulation, currently, a new trend is emerging. This new approach tries to rigorously analyze heuristics. Fiege [19] proposes that if the nature of input is fixed, then heuristics can be theoretically analyzed. Fiege further proposes some rigorous models of input in which one can study heuristics. However, analytic evaluation of heuristics is not totally a new idea. Particularly when the solution space is modeled as a graph, there are several measures which define a good heuristic. Some of these measures are admissibility, informedness and monotonicity (see [41]). Here is a short description:

1. *Admissibility*: Heuristics that find the shortest path to a goal whenever it exists are said to be admissible.

2. *Informedness*: During the traversal of graph representing the solution space, whenever there is a choice among nodes (feasible solutions), the heuristic that selects the node (feasible solution) near to the goal (optimal solution) is said as an informed heuristic. The nearer node the heuristic selects, the more informed it is. Note that this nearness is measured as the path length of the node to goal node,

not the difference between cost of a feasible solution and optimal solution as defined in optimization theory.

3. *Monotonicity*: When a node is reached by the heuristic search, and if there is a guarantee that the same node would not be found later through a shorter path, then such heuristic is said to satisfy monotonicity property.

In the following chapter, a tabu search algorithm is devised for the problem $1 \mid r_j \mid \sum C_j$, modeling the solution space as a graph. But only an empirical analysis of the algorithm is performed, these formal properties of graph search heuristics are not considered. One can refer [41] and [52] for details of graph search heuristics.

Chapter 7

Tabu Search for a Scheduling Problem

In this chapter, two versions of tabu search algorithms are developed for the *NP*-hard scheduling problem $1 \mid r_j \mid \sum C_j$ and their quality are analyzed experimentally. Section 7.1 describes two well-known algorithms for this problem: the Earliest Completion Time (ECT) and Earliest Start Time (EST) heuristics. In Section 7.3, a very inefficient but exact algorithm for this problem is devised. In the remaining sections 7.2, 7.4, 7.5, 7.6 and 7.7, the two tabu search algorithms are developed. Finally in the experimentation part (Section 7.8), the efficiency of these tabu search algorithms is compared against that of ECT and EST heuristics.

7.1 ECT and EST Heuristics

These heuristics are described, for instance, in [7]. Here, these algorithms have been slightly modified to ease in implementation. First consider the ECT heuristic. Let there be n number of jobs denoted as J_j, j = 1 to n. The algorithm goes on building the schedule picking one job at a time. Let at any instant π denote the partial schedule, i.e.,

$$\pi = \pi_1\pi_2\ldots.\pi_k$$

where $k \leq n$. Now, consider the following modification rule.

Rule 7.1 Modification rule for start and complete times

$$s_j = \begin{cases} r_j & \text{if } j = \pi_1 \\ \max\{ r_j, C_{\pi[j-1]}\} & \text{if } j \text{ is in } \pi \\ \text{Max}\{ r_j, C_{\pi[\text{last}]}\} & \text{otherwise } [\pi[\text{last}] \text{ is last job in } \pi]. \end{cases}$$

$$C_j = s_j + p_j$$

Recall that r_j, s_j, p_j and C_j denote release date, start time, processing time and completion time of a job J_j.

Now, the ECT heuristic is presented:

```
**Algorithm** (see [7]) *ECT*
**Begin**
    $\pi$ := empty sequence
    **for** all job $J_j$ **do**            //initialization
        $s_j := r_j$  and $C_j := s_j + p_j$
    unmark all jobs
     **for** count = 1 to n do
        **begin**
            $J_j$ := unmarked job with minimum completion time
            Mark $J_j$
            Append $J_j$ in $\pi$
            **For** all jobs $J_i$ apply rule 7.1
        **End**
    **return** $\pi$
**End of algorithm**
```

Example 7.1 Consider five jobs $J_1,\ldots,J_5$ with the following information:

j	1	2	3	4	5
p_j	11	27	14	21	17
r_j	96	44	26	63	34

Application of the ECT algorithm will give the schedule (3, 5, 2, 4, 1) with total cost 402.

The EST heuristic is a slight modification of ECT. In each iteration of the for-loop, instead of selecting job with minimum completion time, the job with minimum start time is selected. The EST algorithm is not presented here because rest of the algorithm is exactly the same as for ECT.

Time complexity of both ECT and EST algorithms is $\Theta(n^2)$, this can be easily seen.

7.2 Representation of Solution and Solution Space

A solution and the solution space is represented in this way: A feasible solution is represented by a sequence of n numbers from (1, 2, .., n), where n is the number of jobs. Let π be such a sequence. More specifically, let $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, then, $\pi_k = j$ means job J_j is in k'th position of the schedule. Since there are no precedence constraints, any permutation of π will still be a feasible solution, the only problem is, an arbitrary permutation may insert idle times in the schedule. Clearly there are n! number of permutations of π, hence, the solution space will be a graph having n! number of nodes.

Cost of the feasible solution π is denoted by cost(π). Since the objective function is $\sum Cj$, clearly, cost(π) = $\sum_j C_{\pi j}$.

7.3 An Exact Algorithm for 1 | r_j | ģC_j

This algorithm, though very inefficient, can be used for small instances of the problem. Assume that there are n jobs.

In the following algorithm, a subroutine next_permutation(π) has been used.. Given a sequence π, this subroutine generates permutation of π in lexicographic ordering. For example, next permutation of (1, 2, 3, 4, 5) is (1, 2, 3, 5, 4), that of (1, 2, 3, 5, 4) is (1, 2, 5, 3, 4), and so on. For details of this permutation generator, see [51].

The following algorithm is formulated here.

```
Algorithm Exact-algorithm-for- 1 | $r_j$ | ġ$C_j$
Begin
    $\pi$ :=  (1, 2,…., n)
    best_$\pi$ := $\pi$
    best_cost := cost($\pi$)
    for count := 2 to n! do
        begin
            $\pi$ := next_permutation($\pi$)
            if cost($\pi$) < best_cost
                best_$\pi$ = $\pi$
                best_cost = cost($\pi$)
        end
    return best_$\pi$
End of algorithm
```

Even if it is assumed that permutation generation and evaluation of cost both can be done in constant time, the time complexity of this algorithm is still Θ(n!), which is too high.

7.4 Neighborhood Structure for Tabu Search

Now a neighborhood structure is defined for implementing tabu search. For clarity, call this neighborhood as 'swap neighborhood'. Let $\pi = (\pi_1, \pi_2, …., \pi_n)$ be a feasible solution for the problem $1 \mid r_j \mid \sum C_j$, where n is the number of jobs. Then the *swap-neighborhood* of π, $N_s(\pi)$, is defined as

$$N_s(\pi) = \{\pi' : \pi' \text{ is obtained by swapping two jobs in } \pi\}$$

Clearly, $|N_s(\pi)| = {}^nC_2$.

For example, let $\pi = (1, 2, 3)$. Then $N_s(\pi) = \{(2, 1, 3), (3, 2, 1), (1, 3, 2)\}$.

7.5 Tabu List Structure

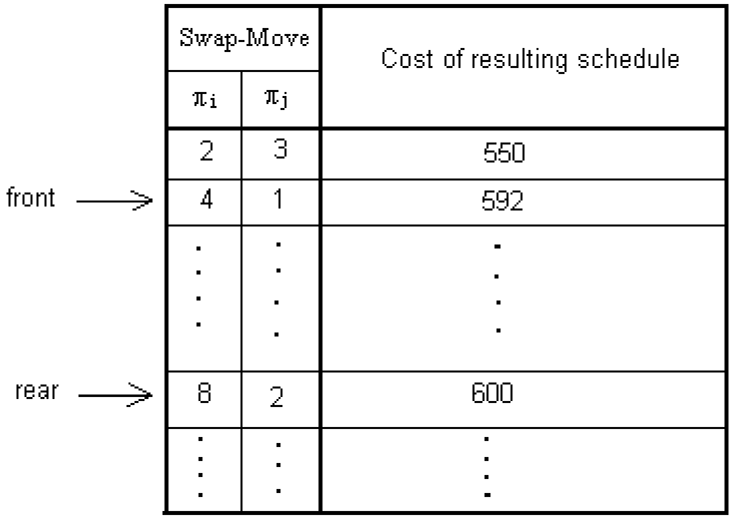
In this section, a tabu list structure is designed for the problem under consideration. As discussed in Section 4.2.2, tabu list stores moves instead of actual solutions. Before defining a move for the tabu search algorithm, first define a ‘swap-move’: Let $\pi = (\pi_1, \pi_2, ..., \pi_n)$ be a feasible solution for the problem $1 \mid r_j \mid \sum C_j$, where n is the number of jobs. Let $\pi' \in N_s(\pi)$ be obtained from π by swapping π_j and π_k. Then the unordered pair $\{\pi_j, \pi_k\}$ is the *swap-move* which converts π to π'.

For example, let $\pi =$ (1, 2, 3, 4, 5). Then the unordered pair {2, 5} is a swap-move which converts π to π', where π' = (1, 5, 3, 4, 2).

In the definition of swap-move, choice of unordered pair is intentional. For clarification, consider what will happen if ordered pair is used. Let π = (1, 2, 3, 4, 5) and apply the swap-move (2,4), the resulting solution will be π' = (1, 4, 3, 2, 5). Now, note that the ordered pairs (2,4) and (4,2) are different, so we can again apply the swap-move (4,2) to π', resulting in π'' = (1, 2, 3, 4, 5), which is same as π. This means a recently visited solution is revisited, which is against the basics of tabu search. On the other hand, use of unordered pair means the swap-moves {2, 4} and {4, 2} are same, hence {4, 2} will not be allowed.

The simple definition of swap-move may not resemble a solution properly. So a move is defined by embedding the cost of resulting solution in the definition of swap-move in this way : Let π be a feasible solution for the problem $1 \mid r_j \mid \sum C_j$. Let π' be the solution obtained by applying swap-move m to π. Then the unordered pair {m, cost(π')} is the *move* that converts π to π'.

The tabu list is implemented as a circular queue of moves. This is illustrated in the following figure.



	Swap-Move		Cost of resulting schedule
	π_i	π_j	
	2	3	550
front →	4	1	592
	⋮	⋮	⋮
rear →	8	2	600
	⋮	⋮	⋮

Figure 7.1 *Tabu list implemented as circular queue for the problem 1 |* r_j | ġC_j

7.6 The Tabu Search Algorithm

In this section, the two versions of tabu search algorithm for 1 | r_j | $\sum C_j$ are developed. As described in Section 4.2.2, a tabu search algorithm uses the basic tabu search algorithm as a subroutine. Their implementations are described separately, for it makes the complexity analysis easier.

7.6.1 Basic Tabu Search

Initialization of the search is implemented and the aspiration conditions are defined in the following way:

Initialization: Guessing that in the optimal solution jobs with earlier completion time come earlier, take the sequence of jobs in non-decreasing order of completion times as the initial solution. This sorting is achieved using quicksort algorithm.

Aspiration criterion: If a tabu move produces a solution that is better than the best solution found so far, accept such a move.

Here, an outline of the basic tabu search is restated so that analysis becomes easier. The algorithm is:

1. s = initial solution
2. For MAXIT number of times do step 3 to 4
3. select s’, the best neighbor of s which is not tabu or which satisfies the aspiration condition.
4. s = s’
5. update the tabu list

First, consider step 3. Let n be the number of jobs in following discussions. Cost evaluation of a feasible solution takes O(n) time. There are ${}^{n}C_2$ number of neighbors, and note that ${}^{n}C_2 =(n^2-n)/2$, i.e., there are $O(n^2)$ neighbors. Hence complexity of finding best neighbor is $O(n^3)$. Now, since tabu list length is TABUSIZE, checking tabu status takes O(TABUSIZE) time, and complexity of step 3 becomes $O(n^3)$+O(TABUSIZE). But since, in practice, tabu list length is much smaller than number of jobs, overall complexity of step 3 is $O(n^3)$.

Since quicksort is taken for initialization, step 1 takes O(nlogn) time. Tabu list as a circular queue, so, step 5 takes constant time. Step 3 iterates for MAXIT number of times, hence, overall complexity is

$$O(n) + O(MAXIT.n^3)$$
$$= O(MAXIT.n^3).$$

This basic tabu search does not consider best, average or worst cases separately. Hence, complexity of this basic tabu search is $O(MAXIT.n^3)$ for all cases.

7.6.2 Complete Tabu Search

Intensification and diversification are implemented in the following way:

Intensification: Intensification is done by swapping two random jobs from the solution sequence obtained by basic tabu search of Section 5.6.1. This scheme is chosen chiefly because it is easier to implement. However, one can guess that the optimum solution is in the proximity of solution obtained by the basic tabu search.

Diversification: Diversification is done by generating random solution.

The complete tabu search is implemented as follows:

1. Initialize and do basic tabu search
2. For MAX_INTESIFY number of times
   - do intensification
   - do basic tabu search
3. for MAX_DIVERSIFY number of times
   - do diversification
   - do basic tabu search

Assuming that a long list of pseudorandom numbers for 1 to n is given, intensification and diversification take constant time. Now, let MAX_INTENSIFY = MAX_DIVERSIFY = M_{ID}. Then complexity of tabu search becomes

$$O(M_{ID}.MAXIT.n^3)$$

Assuming MAXIT = O(n) and $M_{ID} << n$, which is practical, total complexity becomes $O(n^4)$ for all cases.

7.7 ECT_tabu Algorithm

The second version of tabu search algorithm is as follows:

1. Apply the ECT heuristic

2. Take the solution of ECT algorithm as the initial solution, and perform tabu search.

Since the ECT algorithm takes $\Theta(n^2)$, ECT_tabu also has complexity of $O(n^4)$ for all cases.

7.8 Experiments and Results

In this experimentation part, all of the algorithms mentioned in this chapter were implemented in Turbo C++ version 3.0. The source codes for these programs are given in the Appendix.

The objective of implementing ECT, EST and the exact algorithms, described in section 7.1 and 7.3, is simply to compare their output with the output of the two tabu search algorithms developed in this chapter.

For both tabu search algorithms, the following parameters were set:

length of tabu list = 30.

no. of iterations = 50

7.8.1 Input Data Set

First, a program for generating data set was implemented (see Appendix). This program uses the linear congruential pseudorandom number generator provided by the Turbo C++ library. Using this program, six sets of input data were generated, containing instances for 5, 10, 15, 20, 25, and 30 jobs. Each input data set contains 100 instances. In all instances, processing times are in the range [1, 30] and release times are in the range [1, 100].

7.8.2 Output

Input data set	**Number of jobs**	**Exact Solution**	**ECT**	**EST**	**Tabu**	**ECT_tabu**
1	5	356.42	361.88	364.88	356.57	356.13
2	10	864.31	913.02	955.39	898.73	891.13
3	15	--------	1751.68	2000.81	1823.00	1710.28
4	20	--------	2869.27	3416.20	3079.39	2794.22
5	25	--------	4193.81	5181.75	4650.02	4094.62
6	30	--------	5748.52	7330.81	6506.90	5612.95

Table 5.1 *Average of total completion times given by various algorithms*
(For all input data sets, number of instances = 100, maximum processing time = 30, maximum release date = 100)

As shown in above table, tabu search obtained better results than both ECT and EST for number of jobs not exceeding 10. For larger number of jobs, tabu search obtained results better than EST but worse than ECT. Above table is summarized in the following figure.

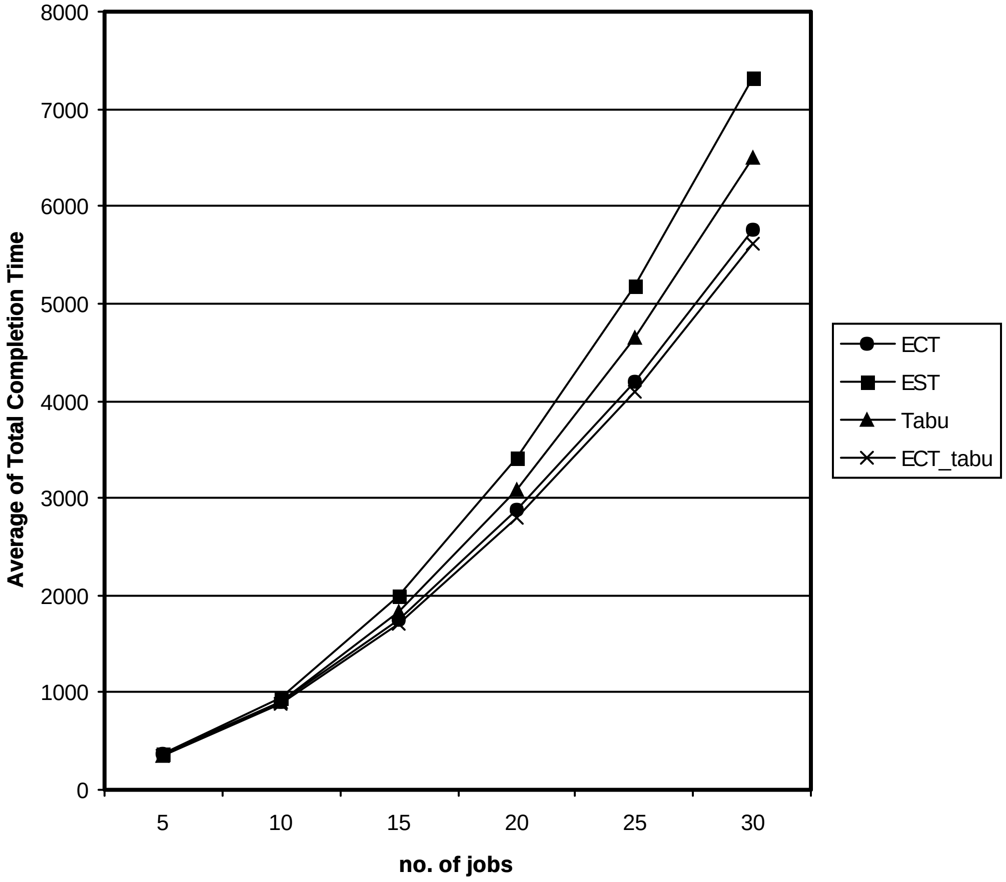


Figure 7.2 *Average of total completion time given by various algorithms*

The experiment did not include instances having number of jobs greater than 30 because the program implementation does not consider overflow errors that may occur while processing large data. These programs can be extended to handle large data by using error-handling mechaninsms provided by Turbo C++. However, it is obvious that ECT_tabu is at least efficient as the ECT heuristic, and once can hope ECT_tabu beats ECT for large instances.

Execution times of all of these programs were in the order of seconds, the detailed description is irrelevant. When excecuted in Intel Pentium IV processor, Windows XP operating system, to process data set of 30 jobs and 100 instances, the ECT_tabu algorithm took 85.18 seconds. This is the largest execution time required in all of these experiments.

Chapter 8

Conclusion and Future Works

In this dissertation, *NP*-hard discrete optimization problems were studied in the context of single machine scheduling. The study also included some polynomially solvable problems and general approaches of tackling *NP*-hard problems, but special focus was given on heuristic evaluation of *NP*-hard single machine scheduling problems. Finally, two versions of tabu search algorithm were devised for the scheduling problem $1 \mid r_j \mid \sum C_j$.

The first version of the tabu search beats both ECT and EST heuristics for instances of 10 jobs or less. But as the number of jobs grows, this tabu search gives solution weaker that ECT, but still better than EST heuristic. The second version of tabu search is much similar to the first version; the only difference is it takes the solution given by ECT as an initial solution. Both versions of tabu search are bounded above by a small polynomial; more specifically, they have complexity of $O(n^4)$ for all cases. This fulfills both of requirements of being a good heuristic, i.e., the heuristic should provide a 'good' solution, and the algorithm should be bounded by a small degree polynomial in the worst case. Thus, in summary, a good tabu search algorithm was implemented for the problem $1 \mid r_j \mid \sum C_j$.

This dissertation could not go into one important area: Categorization of *NP*-hard problems. Knowing *NP*-hardness of a problem is not always pessimistic. There can be *NP*-hard problems whose exact algorithms have complexities higher than polynomial but not *too much*. Such problems are called pseudo-polynomially solvable problems (for example, see [49]). Scheduling problems have also been classified according to hierarchy of such hardness (see [12]). A prospected study is to go into the details of these concepts. Furthermore, as another future study, heuristic algorithms like genetic algorithms can be designed for the problem $1 \mid r_j \mid \sum C_j$ and results compared with the results of tabu search algorithm devised in this dissertation.

Basic Mathematical Notations

Here are the notations used but not defined in this document.

Set theory

$\{a_1, a_2, \ldots, a_n\}$	Set of objects $a_1, a_2, \ldots, a_n$
$\in$	Set inclusion
$A \times B$	Cartesian product of set A and set B
$f: A \times B \rightarrow C$	f is a function from set A×B to set C
2^A	Power set of set A
$\mathbf{N}$	Set of natural numbers
$\Re$	Set of real numbers
$\Re^+$	Set of positive real numbers

Sequence and series

$(a_1, a_2, \ldots., a_n)$	A sequence of numbers $a_1, a_2, \ldots., a_n$
$\sum_j$	Summation over the index j
(a_1, a_2)	Ordered pair
$\{a_1, a_2\}$	Unordered pair

Miscellaneous

$\max\{a, b\}$	Larger number among a and b
$\max\{a_1, a_2, \ldots, a_n)$	Largest among the numbers $a_1, a_2, \ldots, a_n$
x^+	$\max\{x, 0\}$
$n!$	Factorial of integer n
$\{a_1, a_2, \ldots, a_n\}^*$	Kleene closure of finite alphabet $\{a_1, a_2, \ldots, a_n\}$
$[i, j]$	Set of integers $i, i+1, \ldots, j$

References

1. Adolphson, D., and Hu, T. C. (1973): *Optimal Linear Ordering.* SIAM Journal of Applied Mathematics, Vol. 25, pp 403-423.

2. Affenzeller, M., and Mayrhofer, R. (2002): *Generic Heuristics for Combinatorial Optimization Problems.* Proceedings of the 9th International Conference on Operation Research (KOI), pp 83-92.

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

4. Babu, P., Peridy, L., and Pinson, E. (2004): *A Branch and Bound Algorithm to Minimize Total Weighted Tardiness on a Single Processor*. Annals of Operation Research, Vol. 129, pp 33-46.

5. Baker, K. R. (1974): *Introduction to Sequencing and Scheduling.* John Wiley & Sons, NewYork.

6. Baker, K. R., Lawler, E. L., Lenstra, J. K., and Rinnoy Kan, A. H. G. (1983): *Preemptive Scheduling of s Single Machine to Minimize Maximum Cost Subjet to Release Dates and Precedence Constraints.* Operations Research, Vol. 31, no. 2, pp 381-386.

7. Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., and Weglarz, J. (1996): *Scheduling Computer and Manufacturing Processes.* Springer.

8. Brassard, G., and Bratley, P. (1998): *Fundamentals of Algorithmics,* Prentice-Hall of India Pvt. Ltd.

9. Bratley, P., Florain, M., and Robillard, P. (1996): *Scheduling with Earliest Start and Due Date Constraints.* Naval Research Logistics Quarterly, Vol. 18, pp 511-517.

10. Braune, R., Affenzeller, M., and Wagner, S. (2006): *Efficient Heuristic Optimization in Single Machine Scheduling.* http://www.heuristiclab.com/publications

11. Brucker, P. (1995): *Scheduling Algorithms.* Springer.

12. Brucker, P., and Knust, S. (2007): *Complexity Results for Scheduling Problems.* http://www.mathematik.uni-osnabrueck.de/research/OR/class

13. Cook, S. (2003), *The Importance of P versus NP Question,* Journal of the ACM, January, Vol. 50, no. 1, pp. 27-29.

14. Cook, S. (2003): *The P versus NP Problem.* http://www.claymath.org/prizeproblems/pvsnp.htm

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

16. Dhamala, T. N. (2002): *Shop Scheduling Solution Spaces with Algebraic Characterizations.* Otto-von-Geuricke University, Magdeberg, Germany. Ph. D. Thesis (Shaker Verlag, Germany).

17. Dhamala, T.N., and Kubiak, W. (2005): *A Brief Survey of Just-In-Time Sequencing for Mixed Model Production.* International Journal of Operatons Research, Vol.2, pp 38-47.

18. Dhamala, T.N., and Khadka, S. R. (2007): *Just-In-Time Sequencing for Mixed Model Production Systems Revisited.* Discrete Optimization, submitted.

19. Feige, U. (2005): *Rigorous Analysis of Heuristics for NP-Hard Problems.* Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, January.

20. Glover, F., and Laguna (1997): M., *Tabu Search.* http://citeseer.ist.psu.edu/glover97tabu.html.

21. Graham, R. E., Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G. (1979): *Optimization and Approximation in Deterministic Sequencing and Scheduling, A Survey.* Annals of Discrete Mathematics, Vol. 5, pp 287-326.

22. Gupta, M. C., Gupta, Y. P., and Kumar (1993): A., *Genetic Algorithm Application in a Machine Scheduling Problem.* ACM 0-89791-558-5/93/0200/0372.

23. Hariri, A. M. A., and Potts, C.N. (1997): *Single Machine Scheduling with Batch Set-Up Times to Minimize Maximum Lateness.* Annals of Operations Research , Vol. 70, pp 75-92.

24. Hertz, A., Taillard, E., and Werra, D. de (1994): *A Tutorial on Tabu Search.* http://www.cs.colostate.edu/~whitley/cs640/hertz92tutorial.pdf.

25. Hoogeven, J. A., and Vestjens, A. P. A. (1995): *Optimal on-line Algorithms for Single Machine Scheduling.* Memorandum COSOR, 1995, Eindhoven University of Technology, Eindhoven, The Netherlands.

26. Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001): *Introduction to Automata Theory, Languages and Computation.* Pearson Education.

27. Horn, W. A. (1974): *Some Simple Scheduling Algorithms.* Naval Research Logistics Quarterly, Vol. 21, pp 177-185.

28. Jackson, J. R. (1955): *Scheduling a Production Line to Minimize Maximum Tardiness.* Management Science Resource Project, UCLA, Research Report, Vol. 43.

29. Jawor, W. (2005): *Three Dozen Papers on Online Algorithms.* ACM SIGACT News, March, Vol. 36, no. 1.

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

31. Kellerer, H., Tautenhahn, T., and Woeginger, G. J. (1996): *Approximability and Nonapproximability Results for Minimizing Total Flow Time on a Single Machine.* ACM 0-89791-785-5/96/05.

32. Lawler, E. L. (1973): *Optimal Sequencing of a Single Machine Subject to Precedence Constraints.* Management Science, Vol. 19, pp 544-546.

33. Lawler, E. L. (1976): *Sequencing to Minimize the Weighted Number of Tardy Jobs.* RAIRO Operations Research, Vol. 10, pp 27-33.

34. Lawler, E. L. (1978): *Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints.* Annals of Discrete Mathematics, Vol. 2, pp 75-90.

35. Lawler, E. L. (1982): *Sequencing a Single Machine to Minimize the Number of Late Jobs.* Preprint, Computer Science Division, University of California, Berkeley.

36. Lawler, E. L. (1983): *Recent Results in the Theory of Mahicne Scheduling.* In: Bachem, A., Grotschel, M., and Korte, B. (editors), Mathematical Programming: The State of the Art, pp 202-234. Springer, Berlin.

37. Lenstra, J. K., Rinnoy Kan, A. H. G., and Brucker, P. (1977): *Complexity of Machine Scheduling Problem.* Annals of Discrete Mathematics, Vol. 1, pp 343-363.

38. Lenstra, J. K. and Rinnoy Kan, A. H. G. (1979): *Computational Complexity of Discrete Optimization Problems.* Annals of Discrete Mathematics, Vol. 4, pp 121-140.

39. Lewis, H. R., and Papadimitriou, C. H. (1996): *Elements of the Theory of Computation*, Prentice-Hall of India Pvt. Ltd..

40. Liman, S. D., Panwalkar, S. S., and Thongmee, S. (1997): *A Single Machine Scheduling Problem with Common Due Window and Controllable Processing Times.* Annals of Operation Research, Vol. 70, pp 145-154.

41. Luger, G. F. (2001): *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Pearson Education.

42. Mao, W., Kincaid, R. K., and Rifkin A. (1995): *On-line Algorithms for a Single Machine Scheduling Problem.* http://www.citeseer.ist.psu.edu/mao95line.html.

43. Mazure, B., Sais, L., and Gregoire, E. (1997): *Tabu Search for SAT.* American Association for Artificial Intelligence, http://www.aaai.org.

44. McLoughlin III, J.F., and Cedeno, W. (2005): *The Enhanced Evolutionary Tabu Search and Its Application to the Quadratic Assignment Problem.* ACM 1-59593-010-8/05/0006.

45. Milenkovic, M. (1997): *Operating Systems.* Tata McGraw-Hill.

46. Monma, C. L. (1982): *Linear Time Algorithms for Scheduling on Parallel Processors.* Operations Research, Vol. 30, pp 116-124.

47. Moore, J. M. (1968): *A n Job One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs.* Management Science, Vol. 15, pp 102-109.

48. Muth, J. F. and Thompson, G. L. (1963): *Industrial Scheduling.* Prentice Hall, Englewood Cliffs.

49. Papadimitriou, C. H., and Steiglitz, K. (2006): *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall of India Pvt. Ltd.

50. Rangaswamy, B., Jain, A. S., and Glover, F. (1998): *Tabu Search Candidate List Strategies in Scheduling.* 6th INFORMS Advances in Computational and Stochastic Optimization, Logic Programming and Heurisitic Search: Interface in Computer Science and Operations Research Conference in Monterrey Bay, California, January..

51. Rosen, K. H. (2003): *Discrete Mathematics and Its Applications.* Tata McGraw-Hill.

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

53. Sahni, S. (1976): *Algorithms for Scheduling Independent Tasks*. Journal of the ACM, Vol. 23, pp 116-127.

54. Savelsbergh, M. W. P., Uma, R. N., and Wein, J. (1998): *An Experimental Study of LP-Based Approximation for Scheduling Problem.* Proceedings of the ninth annual ACM-SIAM symposium on Discrete Algorithms SODA, January.

55. Taillard, E. (1993): *Benchmark for Scheduling Problems.* European Journal of Operational Research, North Holland, Vol. 64, pp 278-285.

56. Tanenbaum, A. (2004): *Modern Operating Systems*. Prentice-Hall of India Pvt. Ltd.

Appendix

Program Code of Various Algorithms

We give the source code for the algorithms discussed in Chapter 5. The code is in Turbo C++ Version 3.0. Each cpp file contains a complete console based application. Below is a summary of all program files and their details.

S.N.	File name	Page no.	Details
A.1	gdata.cpp	85	File containing code for generating input data
A.2	exact.cpp	86	Source code for the exact algorithm
A.3	ECT.cpp	89	Source code for the ECT heuristic
A.4	Tabu.cpp	92	Source code for tabu search (first version)
A.5	Array.h	97	File containing code for various array manipulations
A.6	Global.h	99	File containing global variables, file handling and miscellaneous functions

Program for the EST heuristic can be obtained by slightly modifying that for ECT heuristic. Similary, program for ECT_tabu can be obtained by combining program codes for ECT heuristic and tabu search, and some minor modifications. We do not include the codes for EST and ECT_tabu due to their redundancy.

A.1 gdata.cpp

```
//generate data set for the problem 1 | rj | SIGMA(cj)
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <stdlib.h>
#include <fstream.h>
#include "d:\thesis\fversion\headers\global.h"
//******************INPUT PARAMETERS**************************

int n_j; //no. of jobs
int pMax;// maximum processing time of a job
int rMax;// maximum release date
int iMax;// number of input data sets to be generated
//************************************************************
char fname[15];

void generate(char c){
        int i, j, k, num, limit;
        char number[5];
        time_t t;
        if(c == 'r') limit = rMax;
        else if(c=='p') limit = pMax;
        else return;
        ofstream file(fname);
```

```
    //write the header
    write_string("No. of jobs = ", file);
    write_num(n_j, file);//number of jobs
    file.put(', ');
    write_string(" No. of instances = ", file);
    write_num(iMax, file);
    if(c == 'r') write_string("\nrelease dates:\n",file);
    else write_string("\nprocessing times:\n", file);
    //now generate the instances of random numbers and save
    srand((unsigned) time(&t));
    for(i=1; i<=iMax; i++){
        for(j=1; j<=n_j; j++){
            num = rand() % limit + 1;
            itoa(num, number, 10);
            for(k = 0; number[k] != '\0'; k++)
                file.put(number[k]);
            file.put(' ');
        }
        file.put('\n');
    }
}


void main(){
    clrscr();
    cout<<"Enter the number of jobs: ";
    cin>>n_j;
    cout<<"\nEnter the number of instances to be generated: ";
    cin>>iMax;
    cout<<"\nEnter the maximum processng time: ";
    cin>>pMax;
    cout<<"\nEnter the maximum release date: ";
    cin>>rMax;
    cout<<"Enter the file to store processing time: ";
    gets(fname);
    generate('p');
    cout<<"\nProcessing times generated.";
    cout<<"\nEnter the file to store release dates: ";
    gets(fname);
    generate('r');
    cout<<"\nRelease dates generated";
    getch();
}
```

A.2 exact.cpp

```
//very inefficient but exact algorithm for 1 | rj | SIGMA(cj)
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include "d:\thesis\fversion\headers\array.h"
#include "d:\thesis\fversion\headers\global.h"

int best_cost;//the current best cost
```

```
long factorial(int n){
    if(n == 1) return 1;
    else return factorial(n-1)*n;
}

void next_permutation(int *a){
    int j = n_jobs -1;
    while(a[j] > a[j+1])
        j = j-1;
    int k = n_jobs;
    while(a[j] > a[k])
        k = k-1;
    swap(a[j], a[k]);
    int r = n_jobs;
    int s = j+1;
    while(r>s){
        swap(a[r], a[s]);
        r = r-1;
        s = s+1;
    }
}

void compute_s_and_c(){//not needed by exact() but for displaying o/p.
    s[pi[1]] = r[pi[1]];
    c[pi[1]] = r[pi[1]] + p[pi[1]];
    for(int i=2; i<=n_jobs; i++){
        s[pi[i]] = max(r[pi[i]], c[pi[i-1]]);
        c[pi[i]] = s[pi[i]] + p[pi[i]];
    }
}


void exact(){
    int temp_pi[MAX];
    for(int i=1; i<=n_jobs; i++)
        pi[i] = i; //an initial soultion to begin
    best_cost = cost(pi);
    copy(pi, temp_pi);
    long max_count = factorial(n_jobs);
    for(long count =2; count <= max_count; count++){
        next_permutation(temp_pi);
        int cst = cost(temp_pi);
        if(cst < best_cost){
            best_cost = cst;
            copy(temp_pi, pi);
        }
    }
    compute_s_and_c();//not needed by exact() but s and c array are useful
}

void adjust_items(int *A, int *x){//adjust the items of x as per the
            //order of items in solution sequnce pi, save the results
            //in A
    for(int j=1; j<=n_jobs; j++)
        A[j] = x[pi[j]];
}
```

```
void main(){
        int no_of_instances;
        int A[MAX], i, j, n_jobs1, n_jobs2, n_i1, n_i2;
        char fname[15], choice = 'n';
        clrscr();
        cout<<"\nVery inefficent but exact solution";
        cout<<" for the problem 1 | rj | SIGMA(cj)";
        cout<<"\nEnter file containing processing times: ";
        gets(fname);
        ifstream pfile(fname);
        if(!pfile)
                error(FILE_OPEN,fname);
        cout<<"\nEnter file containing release dates: ";
        gets(fname);
        ifstream rfile(fname);
        if(!rfile)
                error(FILE_OPEN,fname);
        cout<<"\nEnter file for storing output: ";
        gets(fname);
        ofstream outfile(fname);
        //read headers of the input files and check compatibility
        read_header(pfile, &n_jobs1, &n_i1);
        read_header(rfile, &n_jobs2, &n_i2);
        if( n_jobs1 != n_jobs2 || n_i1 != n_i2)
                error(INCOMPATIBLE_FILES);
        //
        n_jobs = n_jobs1;   no_of_instances = n_i1;
        if(n_jobs > 30) error(MAX_JOB);
        if(n_jobs <= 10){
                cout<<"\nWant to save details? Y/N : ";
                cin>>choice;
        }
        cout<<"\ncomputation in progress, please wait....";
        for(i=1; i<=no_of_instances; i++){
                //read input from data files
                read_row(r, n_jobs, rfile);
                read_row(p, n_jobs, pfile);
                //save input data
                if(choice == 'Y' || choice == 'y') {//save details
                        write_string("INSTANCE ", outfile);
                        write_num(i, outfile);
                        write_string("\nProcessing times:  ", outfile);
                        write_row(p, n_jobs, outfile);
                        write_string("\nRelease dates:     ", outfile);
                        write_row(r, n_jobs, outfile);
                }
                //compute the solution
                exact();
                //save output data
                if(choice == 'Y' || choice == 'y') {//save details
                        write_string("\nOUTPUT DATA:", outfile);
                        write_string("\nSolution sequence: ",outfile);
                        write_row(pi, n_jobs, outfile);
                        write_string("\nProcessing times:  ", outfile);
                        adjust_items(A, p);
```

```
            write_row(A, n_jobs, outfile);
            write_string("\nRelease dates:     ", outfile);
            adjust_items(A, r);
            write_row(A, n_jobs, outfile);
            write_string("\nStart times:       ",outfile);
            adjust_items(A, s);
            write_row(A, n_jobs, outfile);
            write_string("\nCompletion times:  ", outfile);
            adjust_items(A, c);
            write_row(A, n_jobs, outfile);
            write_string("\nCost: ", outfile);
            write_num(best_cost, outfile);
            write_string("\n\n\n", outfile);
        }
        else{//just save complete times
            write_num(best_cost, outfile);
            outfile.put('\n');
        }
    }
    cout<<"\ncomputation completed";
    getch();
}
```

A.3 ECT.cpp

```
//implementing ECT heuristic for the problem 1 | rj | SIGMA(cj)
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include "d:\thesis\fversion\headers\array.h"
#include "d:\thesis\fversion\headers\global.h"

int pi_last_index = 0, //marker to end of sequence in the arry pi
    pi_index =0;//needed by comput_s_c() to know the positon of currently
            //considered job in pi

void insert_in_pi(int j){//insert job j in the current schedule
    pi_last_index++;
    pi[pi_last_index] = j;
    }

int find_in_pi(int j){//find whether job j is in current schedule or not
    for(int i=1; i<=n_jobs; i++)
        if(j == pi[i]){
            pi_index = i;
            return 1;
        }
    return 0;
}

void initialize_s_and_c(){
    for(int i=1; i<=n_jobs; i++){
        s[i] = r[i];
        c[i] = s[i] + p[i];
    }
```

```
}

void compute_s_and_c(){
    for(int i=1; i<=n_jobs; i++){
        if(i == pi[1])
            s[i] = r[i];
        else if(find_in_pi(i))
            s[i] = max(r[i], c[pi[pi_index-1]]);
        else
            s[i] = max(r[i], c[pi[pi_last_index]]);
        c[i] = s[i] + p[i];
    }
}

void copy_finite(int *A,int *B){//copy all values from A to B except infinity
    for(int i=1; i<=n_jobs; i++)
        if(B[i] != infinity) B[i] = A[i];
}

void ECT(){
    int temp_c[MAX];
    for(int i=0; i<=n_jobs; i++) pi[i] = 0;
    for(i=0; i<=n_jobs; i++) temp_c[i] = 0;
    initialize_s_and_c();
    for(i=1; i<=n_jobs; i++){
        //compute_s_and_c();
        copy_finite(c, temp_c);
        int min_index = index_to_min(temp_c, n_jobs);
        temp_c[min_index] = infinity;
        insert_in_pi(min_index);
        compute_s_and_c();
    }
}


void adjust_items(int *A, int *x){//adjust the items of x as per the
            //order of items in solution sequnce pi, save the results
            //in A
    for(int j=1; j<=n_jobs; j++)
        A[j] = x[pi[j]];
}

void main(){
    int no_of_instances;
     //global variable
    int A[MAX], i, j, n_jobs1, n_jobs2, n_i1, n_i2;
    char fname[15], choice = 'n';
    clrscr();
    cout<<"\nUsing ECT heuristic for the problem 1 | rj | SIGMA(cj)";
    cout<<"\nEnter file containing processing times: ";
    gets(fname);
    ifstream pfile(fname);
    if(!pfile)
        error(FILE_OPEN,fname);
    cout<<"\nEnter file containing release dates: ";
    gets(fname);
```

```
ifstream rfile(fname);
if(!rfile)
        error(FILE_OPEN,fname);
cout<<"\nEnter file for storing output: ";
gets(fname);
ofstream outfile(fname);
//read headers of the input files and check compatibility
read_header(pfile, &n_jobs1, &n_i1);
read_header(rfile, &n_jobs2, &n_i2);
if( n_jobs1 != n_jobs2 || n_i1 != n_i2)
        error(INCOMPATIBLE_FILES);
//
if(n_jobs > 30) error(MAX_JOB);
n_jobs = n_jobs1; no_of_instances = n_i1;
if(n_jobs <= 10){
        cout<<"\nWant to save details? Y/N : ";
        cin>>choice;
}
cout<<"\ncomputation in progress, please wait....";
for(i=1; i<=no_of_instances; i++){
        //read input from data files
        read_row(r, n_jobs, rfile);
        read_row(p, n_jobs, pfile);
        //save input data
        if(choice == 'Y' || choice == 'y'){//save details
                write_string("INSTANCE ", outfile);
                write_num(i, outfile);
                write_string("\nProcessing times:  ", outfile);
                write_row(p, n_jobs, outfile);
                write_string("\nRelease dates:     ", outfile);
                write_row(r, n_jobs, outfile);
        }
        //compute the solution
        pi_last_index = pi_index = 0; //global varialbes initialized
        ECT();
        int sum=0;
        for(j=1; j<=n_jobs; j++) sum = sum + c[j];
        //save output data
        if(choice == 'Y' || choice == 'y') {//save details
                write_string("\nOUTPUT DATA:", outfile);
                write_string("\nSolution sequence: ",outfile);
                write_row(pi, n_jobs, outfile);
                write_string("\nProcessing times:  ", outfile);
                adjust_items(A, p);
                write_row(A, n_jobs, outfile);
                write_string("\nRelease dates:     ", outfile);
                adjust_items(A, r);
                write_row(A, n_jobs, outfile);
                write_string("\nStart times:       ", outfile);
                adjust_items(A, s);
                write_row(A, n_jobs, outfile);
                write_string("\nCompletion times:  ", outfile);
                adjust_items(A, c);
                write_row(A, n_jobs, outfile);
                write_string("\nCost: ", outfile);
                write_num(sum, outfile);
                write_string("\n\n\n", outfile);
            }
            else{// just save complete times
                write_num(sum, outfile);
                outfile.put('\n');
            }
        }
        cout<<"\ncomputation completed";
        getch();
}
```

A.4 tabu.cpp

```
//implementing tabu search for 1 | rj | SIGMA(cj)
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include "d:\thesis\fversion\headers\array.h"
#include "d:\thesis\fversion\headers\global.h"

/***************************************************************
                IMPLEMENTING TABU LIST
***************************************************************/
#define MAX_TABUSIZE 30
class Move{
        public:
            int x, y;
            int cost; //cost of the new solution after a move is applied
            Move():x(0),y(0){};
            int operator==(const Move& m);
};

int Move::operator==(const Move& m){
        if((m.x == x) && (m.y == y))
            if(cost == m.cost) return 1;
        return 0;
}



class TabuList{
        private://implemented as a cirular queue
            Move move[MAX_TABUSIZE+1];
            int rear, front, tabusize;
        public:
            TabuList():rear(0),front(0),tabusize(n_jobs) {};
            void initialize();
            void insert(Move m);
            int find(Move m);//returns 1 on success else 0
};

void TabuList::insert(Move m){
        move[rear].x = m.x;
        move[rear].y = m.y;
        if(rear == tabusize - 1)
                rear = 0;
        else rear = rear + 1;
}

int TabuList::find(Move m){
        for(int i=0; i<tabusize; i++)
                if(move[i] == m)
                        return 1;
        return 0;
}

void TabuList::initialize(){
        front = rear = 0;
        for(int i=0; i<tabusize; i++)
                move[i].x = move[i].y = move[i].cost = 0;;
}
//********************************************************************
/********************************************************************
                IMPLEMENTING TABU SEARCH
*********************************************************************/
int best_cost;//cost of the current best sequence

void initialize(){//generate an initial feasible solution in ascending order
        int temp_rp[MAX];// of rj + pj
        temp_rp[0] = infinity;
        for(int i=1; i<=n_jobs; i++)
                temp_rp[i] = r[i] + p[i];
        for(i=1; i<=n_jobs; i++){//create pi order of r[i]+p[i]
                int min_index = index_to_min(temp_rp, n_jobs);
                pi[i] = min_index;
                temp_rp[min_index] = infinity;
        }
        //now compute the corresponding start and complete times
        s[pi[1]] = r[pi[1]];
        c[pi[1]] = s[pi[1]] + p[pi[1]];
        best_cost = c[pi[1]];
        for(i=2; i<=n_jobs; i++){
                s[pi[i]] = max(c[pi[i-1]], r[pi[i]]);
                c[pi[i]] = s[pi[i]] + p[pi[i]];
                best_cost = best_cost + c[pi[i]];
        }
}

TabuList tList;
Move best_move; //move leading to the best neighbor
int best_negh_cost; //cost of the best neighbour
int best_negh[MAX]; //the best neighbor

void generate_neighborhood(){
        best_negh_cost = infinity;
        for(int i=1; i<=n_jobs; i++){
                for(int j=i+1; j<=n_jobs; j++){
                        //best_move is the probable move
                        best_move.x = pi[i];
                        best_move.y = pi[j];
```

```
            copy(pi, best_negh);
            //now create a to-be neighbor by swapping p[i] with p[j]
            swap(best_negh[i], best_negh[j]);
            int cst = cost(best_negh);
            if(cst < best_negh_cost)
                best_negh_cost = cst;
            best_move.cost = cst;
            if(tList.find(best_move))
                if(best_negh_cost >=best_cost)/*if best_move is tabu and it does
                not satisfy the aspiration condition, skip this move*/
                    continue;
        }
    }
}

#define MAXIT 50  //iterations for the tabu search

//basic tabu search: without diversification and intensification
void tabu_search(){
    best_cost = cost(pi);//global variable initialized
    tList.initialize();
    randomize();//initialize the random number generator
    for(int count =1; count<= MAXIT; count++){
        generate_neighborhood();//find cost of the best neighbor
        if(best_negh_cost < best_cost){
            best_cost = best_negh_cost;
            copy(best_negh, pi);
            tList.insert(best_move);
        }
    }
}


void random_swap(){//randomly swap two jobs in pi
    int i, j;
    i = random(n_jobs)+1;
    do{
        j = random(n_jobs)+1;
    }while(j != i);
    swap(pi[i], pi[j]);
}

void random_solution(){//randomly create a feasible solutin pi
    int i, j;
    for(i =1; i<=n_jobs; i++){
        do{
            j = random(n_jobs)+1;
        }while(search(pi,j,i-1));//j is not in in pi[1],pi[2],..,pi[i-1]
        pi[i] = j;
    }
}



#define MAX_INTENSIFY 10
#define MAX_DIVERSIFY 10
```

```
int overall_best_cost, overall_pi[MAX];


void compute_s_and_c(){//not needed by full_tabu_serach() but for displaying o/p
        s[overall_pi[1]] = r[overall_pi[1]];
        c[overall_pi[1]] = r[overall_pi[1]] + p[overall_pi[1]];
        for(int i=2; i<=n_jobs; i++){
                s[overall_pi[i]] = max(r[overall_pi[i]], c[overall_pi[i-1]]);
                c[overall_pi[i]] = s[overall_pi[i]] + p[overall_pi[i]];
        }
}

//full tabu search: including diversification and intensification
void full_tabu_search(){
        int temp_pi[MAX];
        initialize(); //chose an initial solution pi
        tabu_search();
        overall_best_cost = best_cost;
        copy(pi, overall_pi);
        //intensify: begin search in the proximity of best solution
        for(int i=1; i<=MAX_INTENSIFY; i++){
                tabu_search();
                if(best_cost < overall_best_cost){
                        overall_best_cost = best_cost;
                        copy(pi, overall_pi);
                }
                random_swap();//slightly change the best solution;
        }

        for(i = 1; i<=MAX_DIVERSIFY; i++){
                tabu_search();
                if(best_cost < overall_best_cost){
                        overall_best_cost = best_cost;
                        copy(pi, overall_pi);
                }
                random_solution();
        }
        compute_s_and_c();//not needed by full_tabu_search() but for displaying
}


void adjust_items(int *A, int *x){//adjust the items of x as per the
                        //order of items in solution sequnce pi, save the results
                        //in A
        for(int j=1; j<=n_jobs; j++)
                A[j] = x[overall_pi[j]];
}

void main(){
        int no_of_instances;
        int A[MAX], i, j, n_jobs1, n_jobs2, n_i1, n_i2;
        char fname[15], choice = 'n';
        clrscr();
        cout<<"\nUsing Tabu Search for the problem 1 | rj | SIGMA(cj)";
        cout<<"\nEnter file containing processing times: ";
        gets(fname);
```

```
ifstream pfile(fname);
if(!pfile)
	error(FILE_OPEN,fname);
cout<<"\nEnter file containing release dates: ";
gets(fname);
ifstream rfile(fname);
if(!rfile)
	error(FILE_OPEN,fname);
cout<<"\nEnter file for storing output: ";
gets(fname);
ofstream outfile(fname);
//read headers of the input files and check compatibility
read_header(pfile, &n_jobs1, &n_i1);
read_header(rfile, &n_jobs2, &n_i2);
if( n_jobs1 != n_jobs2 || n_i1 != n_i2)
	error(INCOMPATIBLE_FILES);
//
n_jobs = n_jobs1;  no_of_instances = n_i1;
if(n_jobs > 30) error(MAX_JOB);
if(n_jobs <= 10){
	cout<<"\nWant to save details? Y/N : ";
	cin>>choice;
}
cout<<"\ncomputation in progress, please wait....";
for(i=1; i<=no_of_instances; i++){
	//read input from data files
	read_row(r, n_jobs, rfile);
	read_row(p, n_jobs, pfile);
	//save input data
	if(choice == 'Y'|| choice == 'y') {//save details
		write_string("INSTANCE ", outfile);
		write_num(i, outfile);
		write_string("\nProcessing times:  ", outfile);
		write_row(p, n_jobs, outfile);
		write_string("\nRelease dates:     ", outfile);
		write_row(r, n_jobs, outfile);
	}
	//compute the solution
	full_tabu_search();
	//save output data
	if(choice == 'Y' || choice == 'y') {//save details
		write_string("\nOUTPUT DATA:", outfile);
		write_string("\nSolution sequence: ",outfile);
		write_row(overall_pi, n_jobs, outfile);
		write_string("\nProcessing times:  ", outfile);
		adjust_items(A, p);
		write_row(A, n_jobs, outfile);
		write_string("\nRelease dates:     ", outfile);
		adjust_items(A, r);
		write_row(A, n_jobs, outfile);
		write_string("\nStart times:       ",outfile);
		adjust_items(A, s);
		write_row(A, n_jobs, outfile);
		write_string("\nCompletion times:  ", outfile);
		adjust_items(A, c);
		write_row(A, n_jobs, outfile);
```

```
                        write_string("\nCost: ", outfile);
                        write_num(overall_best_cost, outfile);
                        write_string("\n\n\n", outfile);
                }
                else{//just save complete times
                        write_num(overall_best_cost, outfile);
                        outfile.put('\n');
                }
        }
        cout<<"\ncomputation completed";
        getch();
}
```

A.5 array.h

```
/*CONTAINS
        quicksort(A, n) :- A is the array and n its size
        int search(A, Asize, key) :- peform linear search
        T max(A, ASize)
        T min(A, ASize)
        T max(x, y)
        T min(x, y)
*/

#ifndef SORT_H_
#define SORT_H_

#include <assert.h>

template <class T>
void swap(T& x, T& y){
        T temp = x;
        x = y;
        y = temp;
}

template <class T>
void quicksort(T data[], int first, int last){
        int lower = first +1, upper = last;
        swap(data[first], data[(first+last)/2]);
        T bound = data[first];
        while(lower <= upper){
                while(data[lower] < bound)
                        lower++;
                while(bound<data[upper])
                        upper--;
                if(lower<upper)
                        swap(data[lower++],data[upper--]);
                else lower++;
        }
        swap(data[upper],data[first]);
        if(first<upper-1)
```

```
                quicksort(data,first,upper-1);
        if(upper+1<last)
                quicksort(data,upper+1,last);
}

template <class T>
void quicksort(T data[], int n){
        if(n<2) return;
        for(int i=1, max=0; i<n; i++)
                if(data[max]<data[i])
                        max = i;
        swap(data[n-1],data[max]);
        quicksort(data,0,n-2);
}

//linear search: return 1 on sucess, 0 on failure
template <class T>
int search(T A[], T key, int ASize){
        for(int i=1; i<=ASize; i++)
                if(A[i] == key) return 1;
        return 0;
}

template <class T>
T max(T A[], int ASize){
        int tmax = A[1], i;
        for(i=1; i<=ASize; i++)
                if(tmax<A[i]) tmax = A[i];
        return tmax;
}

template <class T>
T min(T A[], int ASize){
        int tmin = A[1], i;
        for(i=1; i<=ASize; i++)
                if(tmin>A[i]) tmin = A[i];
        return tmin;
}

template <class T>
int index_to_min(T A[], int ASize){
        int min_index = 1, i;
        for(i=1; i<=ASize; i++)
                if(A[min_index] > A[i] ) min_index = i;
        return min_index;
}

template <class T>
T max(T x, T y){
        if(x>y) return x;
        else return y;
}

template <class T>
T min(T x, T y){
        if(x<y) return x;
```

```
		else return y;
}


/*
template <class T>
void create_array(T *A, int ASize){
		A = new T[ASize];
		assert(A!=0);
}


template <class T>
void create_array(T **A, int nRows, int nCols){
		A = new T*[nCols];
		assert(A!=0);
		for(int i=0; i<nCols; i++){
				A[i] = new T[nRows];
				assert(A[i]!=0);
		}
}
*/
#endif
```

A.6 global.h

```
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "d:\thesis\fversion\headers\array.h"

#define MAX 31 //maximum of 30 jobs
#define infinity 9999 //maximum integer.


int n_jobs; //no. of jobs
//int *p, *r, *s, *c, *pi;//pointers to array of processing times, release times,
						//start times and completion times, respectively.

int p[MAX], r[MAX], s[MAX], c[MAX], pi[MAX];

enum ERROR_TYPE { FILE_OPEN, INCOMPATIBLE_FILES, MAX_JOB, UNKNOWN};

void error(ERROR_TYPE err, char *string = 0){
		switch(err){
				case FILE_OPEN:
						cout<<"\nCould not open "<<string;
						break;
				case INCOMPATIBLE_FILES:
						cout<<"\ndata files containg p and r times are incompatible";
						cout<<"\nno. of jobs and/or no. of instnances do not match";
						break;
				case MAX_JOB:
            cout<<"\nMaximum allowable number of jobs is 30.";
            break;
        default:
            cout<<"\nFatal error. ";
    }
    getch();
    exit(0);
}

void copy(int *A, int *B){//copy contents of A into B
    for(int i=1; i<=n_jobs; i++)
        B[i] = A[i];
}

int cost(int *seq){//gives cost of a sequence
    int cst, s_time;
    s_time = r[seq[1]];
    cst = s_time + p[seq[1]];
    for(int i=2; i<=n_jobs; i++){
        s_time = max(s_time + p[seq[i-1]],r[seq[i]]);
        cst = cst + s_time + p[seq[i]];
    }
    return cst;
}

int odd(int x){
    return (x/2 == (x+1)/2);
}

void read_header(ifstream& file, int *num_jobs, int *n_instances){
    int i, j;
    char ch, numstring[5];
    for(i=1; i<=2; i++){
        ch = file.get();
        while(!isdigit(ch))
            ch = file.get();
        j=0;
        numstring[j++] = ch;
        ch = file.get();
        while(isdigit(ch)){
            numstring[j++] = ch;
            ch = file.get();
        }
        numstring[j] = '\0';
        if(i==1)
            *num_jobs = atoi(numstring);
        else *n_instances = atoi(numstring);
    }
}

void read_row(int *A, int row_length, ifstream& file){
    char ch, numstring[5];
    int i, j;
    for(i=1; i<=row_length; i++){
        ch = file.get();
        while(!isdigit(ch))
```

```
            ch = file.get();
        j=0;
        numstring[j++] = ch;
        ch = file.get();
        while(isdigit(ch)){
            numstring[j++] = ch;
            ch = file.get();
        }
        numstring[j] = '\0';
        A[i] = atoi(numstring);
    }
}

void write_row(int *A, int row_length, ofstream& file){
    int i, j, k, len;
    char numstring[5];
    for(i=1; i<=row_length; i++){
        itoa(A[i], numstring, 10);
        j=0;
        while(numstring[j] != '\0')
            file.put(numstring[j++]);
        for(k = j; k<=7; k++)//do format by inserting blanks
            file.put(' ');
    }
}

void write_string(char *string, ofstream& file){
    int i=0;
    while(string[i] != '\0')
        file.put(string[i++]);
}

int read_num(ifstream& file){
    char ch, numstring[5];
    int j;
    ch = file.get();
    while(!isdigit(ch))
        ch = file.get();
    j=0;
    numstring[j++] = ch;
    ch = file.get();
    while(isdigit(ch)){
        numstring[j++] = ch;
        ch = file.get();
    }
    numstring[j] = '\0';
    return atoi(numstring);
}

void write_num(int num, ofstream& file){
    char numstring[5];
    itoa(num, numstring, 10);
    int i=0;
    while(numstring[i] != '\0')
        file.put(numstring[i++]);
}
```