Chapter 3 General-Purpose Scheduling Procedures

3.1 Introduction

Some scheduling problems are inherently easy.

Many of these can be formulated as linear programs that can be solved by efficient algorithms.

The efficient algorithms are usually referred as **polynomial time algorithms**.

Even large problem can be solved in a **relatively short time**.

However, many more scheduling problems are intrinsically very hard.

The problems are called **NP-hard**. They cannot be formulated as linear programs, and no simple rules or algorithms yield optimal solutions in a limited amount of computer time.

It may be formulated as **integer** or **disjunctive programs**.

Solving them to optimality may require an enormous amount of computer time.

One is usually satisfied with an “acceptable” feasible solution that presumably is not far from optimal.

This chapter presents a number of **general-purpose techniques** that have been proven useful in industrial scheduling.

Most of these methods typically do not guarantee an optimal solution; their goal is to find a reasonably good solution in a relatively short time.

Basic dispatching rule

Combining basic dispatching rule into **hybrid or composite rules**

Branch and bound techniques are based on complete enumeration. The only technique discussed in this chapter designed to find an optimal solution. However, substantial amount of computer time.

Beam search, eliminate search alternatives in an intelligent way. Does not guarantee an optimal solution.

Local search techniques:

Simulated annealing,

Tabu-search,

Genetic algorithms

These fairly generic techniques can be applied to different scheduling problem with only minor customization.

3.2 Basic Dispatching Rules

Dispatching rule: **prioritizes** all the jobs that are waiting for processing.

This scheme takes into account the jobs' attributes, the machines' attributes, as well as current time.

Research for several decades: many rules have been developed and studies in the literature.

Classified in various ways: **static** rule and **dynamic** rule.

Static rules are **not time-dependent**. They are just a function of the job data, the machine data, or both.

Dynamic rules are **time-dependent**.

EG/minimum slack first (MS) rule orders jobs according to their remaining slack.

Remaining slack are defined as $\max(d_j - p_j - t, 0)$, where t is current time.

A second way of classifying: **local** rule and **global** rule.

Local rule use only information pertaining to current machine and current queue.

Global rule use information pertaining to other machines.

EG/ processing time of the job on the next machine.

Sample of Basic dispatching rules:

Service in random order (SIRO) rule: random selection.

Earliest release date first (ERD) rule: equivalent to first-come-first-served rule. This rule in a sense minimizes the variation in the waiting times.

Earliest due date first (EDD) rule: This rule tends to minimize the maximum lateness among the jobs. In single machine, with n jobs available at time 0, the EDD does minimize the maximum lateness.

Minimum slack first (MS) rule: a variation of EDD rule. This rule tends to minimize due date related objectives.

Weighted shortest processing time first (WSPT) rule: When a machine is free, the job with the highest ratio of weight over processing time (w_j / p_j) is scheduled next.

This rule tends to minimize the weighted sum of the completion times ($\sum w_j C_j$).

In single machine, with n jobs available at time 0, the WSPT does minimize $\sum w_j C_j$.

When all weights are equal, the WSPT reduced to SPT rule (shortest processing time).

Longest processing time first (LPT) rule: When there are machine in parallel, this rule tends to **balance** the workload over the machine.

Keep short processing time job for balance at later stage to balance machine loading.

Shortest setup time first (SST) rule.

Least flexible job first (LFJ) rule: used when there are a number of non-identical machines in parallel and jobs are subject to machine-eligibility constraints.

Job j can only processed by set M_j.

When machine is free, job with fewest processing alternatives is selected.

Critical path (CP) rule: used when jobs subject to precedence constraints. It selects the job on the critical path (longest string of processing times in the precedence-constraints graph).

Largest number of successors (LNS) rules: used when jobs subject to precedence constraints. It selects the job with largest number of followers.

Shortest queue at the next operation (SQNO) rule: used in job shop. The job with shortest queue at the next operation on its route is selected for processing. The queue size may either be the **number of waiting jobs** or **total amount of work**.

Table 3.1. Basic dispatching rule and their objectives.

The basic dispatching rules described in Table 3.1 are of limited use.

When a complex objective has to be minimized, none of the basic dispatching rules may perform effectively.

3.3 Composite Dispatching Rules

Basic dispatching rules are useful for a single objective, such as makespan, sum of completion times, or maximum lateness.

A realistic objective may be a combination of several basic objectives.

Sorting the jobs on the basis of one or two parameters may not lead to acceptable schedules.

General framework of composite dispatching rule:

A **composite dispatching rule** is a **ranking expression** that combines a number of basic dispatching rules.

An **attribute** is any property associated with jobs or machines; it may constant or time-dependent.

EG/ job attributes are weight, processing time, due date.

EG/ machine attributes are speed, the number of jobs waiting.

Scaling parameter scale the contribution of basic rules to the total ranking expression.

They may be fixed or may be variables and a function of the job set.

If the scaling parameters depend on the particular **job set**, they require the computation of some **job set statistics** (***factor***) that characterize the scheduling instance as accurately as possible.

EG/ whether or not the due dates of the jobs are tight.

The function that maps the statistics into the scaling parameters has to be determined by the designer of the rule.

Experience or extensive computer simulations are required.

Each time the composite dispatching rule is used for generating a schedule, the necessary statistics are computed. Based on the values of these statistics, the values of the scaling parameters are set by the predetermined functions. After the scaling parameters are set, the dispatching rule is applied to the job set.

Example problem: single machine, *n* jobs (all available at time 0). Weighted tardiness ($\sum w_j T_j$) as the objective to be minimized.

This problem is very hard and no efficient algorithm is known.

Some heuristics:

(1) WSPT: optimal when all release dates and due dates are zero.

(2) EDD or MS: optimal when all due dates are sufficiently loose and spread out.

It is natural to combine these rules.

Apparent Tardiness Cost **(ATC):** combines WSPT and MS.

When machine free, a ranking index is computed for each job.

$$I_j(t) = \frac{w_j}{p_j} \exp\left(-\frac{\max(d_j - p_j - t, 0)}{K\,\overline{p}} \right)$$

K is the scaling parameter, determined empirically

$\overline{p}$:average processing time

If *K* is very large, the rule is reduced to WSPT.

If *K* is very small and there are no overdue jobs, the rule reduced to MS.

If *K* is very small and there are overdue jobs, the rule reduced to WSPT applied to the overdue jobs.

To have good schedule, the value of *K* (***look-ahead parameter***) must be appropriate for the particular instance of the problem.

Performing statistical analysis:

Due date tightness factor $\tau = 1 - \frac{\overline{d}}{C_{\max}}$, $\overline{d}$ is the average due date.

Close to 1: due dates are **tight**. Close to 0: due dates are **loose**.

Due date range factor $R = \frac{d_{\max} - d_{\min}}{C_{\max}}$.

A high value for *R* implies large range for due dates.

A significant amount of research has been done to establish the relationship between the scaling parameter *K* and the factor τ and *R*.

<u>Steps:</u>

First, characterize the problem through these two **statistics**.

Then, determine the *K* as a function of these factors.

After fixing *K*, one applies the rule.

Several generalizations of the ATC rules: release date, sequence-dependent setups.

Apparent tardiness cost with setups (ATCS): *n* jobs; a single machine; sequence-dependent setups s_{jk}; the objective minimize the sum of weighted tardiness.

ATCS combines WSPT, MS, and SST.

Calculate the index of job *j* upon the completion of job *l* at time *t*:

$$I_j(t,l) = \frac{w_j}{p_j} \exp\left(-\frac{\max(d_j - p_j - t, 0)}{K_1 \bar{p}}\right) \exp\left(-\frac{s_{lj}}{K_2 \bar{s}}\right)$$

$\bar{s}$: the average of the setup time.

K_1: the due date related scaling parameter

K_2: the setup related scaling parameter

K_1 and K_2 are function of three factors:

(1) The due date tightness factor τ

(2) The due date range factor *R*

(3) The setup time severity factor $\eta = \bar{s} / \bar{p}$

The makespan (now are schedule dependent) are estimated by

$\hat{C}_{\max} = \sum_{j=1}^{n} p_j + n\bar{s}$ (This is likely to be overestimated.)

An experimental study of the ATCS rule suggested

$K_1 = 4.5 + R \quad \text{for } R \le 0.5$

$K_1 = 6 - 2R \quad \text{for } R \ge 0.5$

$K_2 = \tau / (2\sqrt{\eta})$.

EG 3.3.1 The ATCS rule(p35)

Sequence 2,4,3,1 with total weighted tardiness equal to 98. Complete enumeration shows that this sequence is optimal.

3.4 BRANCH AND BOUND

enumeration schemes

a class of schedule are discarded by showing that (1) the objective values of all these schedules are higher than a **provable lower bound** and (2) this lower bound is higher than (or equal to) the value of the objective of a **feasible schedule** obtained earlier.

Consider a single machine and *n* jobs with different release dates and due dates. The objective is to minimize the maximum lateness, and preemptions are not allowed.

This problem is NP-hard.

It is important because it appears frequently as a subproblem in heuristic procedures for flow shop and job shop scheduling.

A branch and bound procedure:

Branch:

Schedules are developed starting from the beginning of the schedule.

A single node at level 0, which is at the top of the tree.

There are *n* branches going down to *n* nodes at level 1. Specify a job in the first position. There are $n \times (n-1)$ nodes at level 2.

At level *k*, the jobs in the first *k* positions are specified.

Actually, it is often not necessary to consider every remaining job as a candidate for the next position.

Dominance rule:

If jobs $j_1, \ldots\ldots j_{k-1}$ are assigned to the first *k*-1 positions, job *c* has to be considered as a candidate for position *k* only if

$r_c < \min_{l \in J}\left(\max(t, r_l) + p_l\right)$,

where

J: the set of jobs not yet scheduled.

t: the time the machine completes job j_{k-1} and is free to start the next job.

If job *c* does not satisfy this inequality, that is, if

$r_c \geq \min_{l \in J}\left(\max(t, r_l) + p_l\right)$,

then it make sense to put the job minimizes the right hand side in position *k* and job *c* in position *k*+1. This would not affect the completion time of job *c* in any way. So in this case job *c* does not have to be considered for position *k*.

Bound:

An easy lower bound for a node at level *k*-1 can be established by scheduling the remaining jobs *J* according to the *preemptive EDD rule*.

The preemptive EDD rule is optimal for this problem when preemptions are allowed, thus provides a lower bound for the nonpreemptive problem.

If a preemptive EDD rule yields a nonpreemptive schedule, then it is feasible one. (then all nodes with a higher lower bound can be disregarded.)

Example 3.4.1

Level 1:

(3,*,*,*) and (4,*,*,*) are disregarded immediately by dominance rule.

(1,*,*,*) lower bound 5

(2,*,*,*) lower bound 7

Level 2:

(1,2,*,*) lower bound 6, preemptive schedule 1,2,4,3

(1,3,*,*) lower bound 5, nonpreemptive (feasible) schedule 1,3,4,2

(1,4,*,*) lower bound 6. nonpreemptive (feasible) schedule 1,2,4,3

Optimal: 1,3,4,2.

The problem without precedence constraints is a special case of the problem with precedence constraints. The later is at least as hard.

From enumeration point of views, the problem with precedence constraints is actually somewhat easier than that without precedence constraints, as certain schedule that violate precedence constraints can be ruled out immediately.

This reduces the number of schedules that have to be considered. (Less number of feasible schedules).

This nonpreemptive, single-machine scheduling problem with release dates, due dates, and precedence constraints and maximum lateness as the objective is a very important scheduling problem.

Job shop scheduling problems are often decomposed so that in the solution process many single-machine problems of this type have to be solved.

3.5 BEAM SEARCH

Enumerative branch and bound methods are the most used methods for obtaining optimal solution to **NP-hard** scheduling problems. But, it can be extremely time-consuming.

Beam search, a derivative of branch and bound, tries to eliminate branches intelligently. But, can no longer guarantee an optimal solution.

Consider again a single machine problem with n jobs.

At level 1, there are n nodes.

At level k, there are totally $n!/(n-k)!$ nodes.

At the bottom level, there are $n!$ nodes.

By determining the lower bound of the offspring of a node, one may eliminate the node and its offspring.

If one could obtain a reasonable good feasible schedule through some clever heuristic before starting branch and bound, one can eliminate many nodes.

Dominance rules may also reduce the number of nodes.

However, even after these eliminations branch and bound usually still has too many nodes to evaluate.

With beam search only the most promising nodes at level *k* are selected to branch from. The remaining nodes at the level are discarded permanently.

The number of nodes retained is the ***beam width*** of the search.

There is a trade-off: a crude prediction: quick but may discard good solutions. A thorough evaluation may be prohibitively time-consuming.

Two-stage approach is useful.

Filter width: through crude evaluation, the number of nodes remains.

Beam width: the nodes that pass filter are evaluated thoroughly, only the number of beam width are remained in the tree.

EG:

Crude prediction: due date tightness (or some other statistic)

Thorough evaluation: composite dispatching rule. The value of the schedule's objective represents an upper bound on the best schedule among the offspring of that node.

Example 3.5.1 (p.40, simplified version of Beam Search)

Single machine total weighted tardiness problem.

All jobs available at time 0.

No sequence-dependent setup.

No filtering mechanism is used.

Beam width is 2, so that at each level two nodes are remained.

The prediction: sequencing the unscheduled jobs using the ATC rules.

Due date range factor R=11/37

Due date tightness factor τ=32/37

Look-ahead parameter is chosen to be 5

3.6 LOCAL SEARCH: SIMULATED ANNEALING AND TABU-SEARCH

So far all the algorithms described in this chapter have been of **constructive type**: gradually construct a schedule by adding one job at a time.

Improvement type: start with a **complete schedule**, then try to obtain a better schedule by manipulating the current schedule.

An important class of improvement type algorithms are the local search procedure.

A local search does not guarantee an optimal solution.

A neighbor is obtained by a **well-defined modification** of a current schedule.

At each iteration, search procedure performs a search within the neighborhood and evaluates various neighboring solutions.

Based on a given acceptance-rejection criterion, the procedure either accept or reject a candidate solution as the next schedule to move to, based on a given acceptance-rejection criterion.

One can compare the various local search procedures on the following four design criteria:

(i) the schedule representation

(ii) neighborhood design

(iii) search process within the neighborhood

(iv) acceptance-rejection criterion

representation of a schedule may be trivial.

A nonpreemptive single machine schedule can be specified by a permutation of the n jobs.

A nonpreemptive job shop can be specified by m consecutive strings, each one representing a permutation of n operations on a specific machine.

Based these, the staring and completion times of all operations can be computed.

When preemption are allowed, the representation becomes significantly more complicated.

The design of neighborhood is very important.

For a single machine a neighborhood of a particular schedule can be simply performing a single adjacent pairwise interchange.

n-1 neighboring schedule of a current schedule.

A larger neighborhood: Taking an arbitrary job in the schedule and insert in n-1 other positions. Less than $n(n-1)$ neighbors, as some of them are identical.

An interesting example: neighbor design for job shop with makespan as objective.

A critical path consists of a set of operations of which the first one starts out at time t=0 and the last one finishes at time t= $C_{\max}$.

The completion of each operation on a critical path is equal to the starting time of the next operation on that path.

Two successive operations either belong to the same job or are processed on the same machine.

A schedule may have multiple critical path.

To reduce makespan, changes have to be made in the sequence of the operations on the critical path.

To interchange a pair of adjacent operations on the critical path, the operations must be on the same machine and belong to different jobs.

In single critical path, the number of neighbors is the number of operation on the path minus 1.

Experiments have shown that this type of neighborhood for the job shop problem is too simple to be effective.

More sophisticated neighborhoods have been designed for better preformance.

One of these is the ***one step look-back interchange***.

Example 3.6.1

Figure 3.4

(a) current schedule

(b) interchange (i,j) and (i,k)

(c) interchange (h, l) and (h,k).

multi-step look-back interchanges

The search process within a neighborhood can be done in a number of ways.

It may pay to do a more organized search and then select first schedules that appear promising.

Acceptance-rejection criterion is the design aspect that distinguishes a local search procedure the most.

Simulated annealing: probabilistic process.

Tabu-search: deterministic process.

<u>**Simulated Annealing:**</u>

Origin in the fields of material science and physics.

Physical annealing process of condensed matter

At iteration *k*, a current schedule S_k and a best schedule found so far S_0.

$G(S_k) \geq G(S_0)$

$G(S_0)$ is often termed the *aspiration criterion*.

At iteration k, search is conducted within the neighborhood of S_k.

First, a candidate schedule S_c is selected.

The selection of a candidate can be random or in an organized way.

If $G(S_c) < G(S_k)$, a move is made by setting $S_{k+1} = S_c$.

If $G(S_c) < G(S_0)$, $S_0 = S_c$.

If $G(S_c) \geq G(S_k)$, a move is made to S_c with probability

$$P(S_k, S_c) = \exp\left(\frac{G(S_k) - G(S_c)}{\beta_k}\right).$$

With probability $1 - P(S_k, S_c)$, schedule S_c is rejected, setting $S_{k+1} = S_k$.

The $\beta_1 \geq \beta_2 \geq \beta_3 \geq ... \geq 0$ are control parameters refered to as cooling parameter or temperatures.

Often, $\beta_k = a^k$, where a is between 0 and 1.

Move to worse solutions are allowed. Give the procedure the opportunity to move away from a local minimum.

The acceptance probability for non-improving move is lower in later iterations.

If a neighbor is significantly worse, its acceptance probability is very low and the move is unlikely to be made.

Stopping criteria: a pre-specified number of iterations. Run until no improvement has been obtained for a given number of iterations.

Algorithm 3.6.1. Simulated Annealing (p45)

The effectiveness depends on the design of the neighborhood and how the search is conducted within the neighborhood.

Key:

(1) Facilitate moves to better solutions

(2) move out of local minima.

Over the last two decades, considerable success.

<u>Tabu-search:</u>

The search for a neighbor within the neighborhood as a potential candidate to move to.

As in simulated annealing, this search can be done randomly or in an organized way.

The basic difference between these two methods lies in the mechanism used for approving a candidate schedule.

A tabu-list of mutations, which the procedure is not allowed to perform, is kept.

The tabu-list has a fixed number of entries (usually between five and nine).

Every time a move is made by a mutation in the current schedule, the reverse mutation is entered at the top of the tabu-list; all other entries are pushed down one position, and the bottom entry is deleted.

The reverse mutation is put on the tabu-list to avoid returning to a local minimum that has been visited before.

If the number of entries in the tabu list is too small, cycling may occur; if the number of too large, the search may be overly constrainted.

Algorithm 3.6.2. Tabu-search (p.46)

Example 3.6.2 single machine total weighted tardiness problem in Example 3.5.1.

Tabu list: the last two moves.

Tabu-tree:

In this tree each node represents a solution or schedule.

While the search process goes from one solution to another (with each solution having a tabu-list), the process generates additional nodes.

Certain solutions that appear promising may not be used as takeoff point immediately but are retained for future.

If at a certain point during the search process current solution does not appear promising, the search process can return to another node.

3.7 LOCAL SEARCH: GENETIC ALGORITHMS

Genetic algorithms are more general and abstract than simulated annealing and tabu-search.

Simulated annealing and tabu-search can be viewed as special case of genetic algorithm.

View schedule as *individuals* or *member* of a *population*.

The fitness of an individual is measured by the associated value of the objective function.

The procedure works iteratively, and each iteration is a generation.

The population of a generation consists of individuals surviving from the previous generation plus the *children* from previous generation.

Population size usually remains fixed.

The children are generated through reproduction and mutation of parents.

Individuals are referred as referred to as "*chromosomes*."

Mutation: pairwise interchange.

At each iteration, a number of schedules are generated and carried over to the next step.

Simulated annealing and tabu search are special case of genetic algorithm with a population size equal to 1.

Crossover effect: combining parts of different schedules.

A very simplified version of a genetic algorithm:

Algorithm 3.7.1. Genetic Algorithm

Advantage: can be applied without knowing much about the structural properties of the problem.

Disadvantage: the amount of computation time needed to obtain good solution can be relatively long.

3.8 DISCUSSION

combines several of the techniques.

The following three steps has proven useful for solving scheduling problem in practice. It combines composite dispatching rules with simulated annealing or tabu-search.

Step 1: compute statistics

Step 2: determine the scaling parameters of a composite dispatching rule and apply the composite dispatching rule to the scheduling instance.

Step 3: Use the schedule developed in step 2 as an initial solution for a local search procedure.

Empirical procedure that determines the functions that map the statistics to the scaling parameters constitutes a major investment.

It may be possible to use composite dispatching rules within the beam search process in order to evaluate nodes.

Also, a final solution obtained with beam search can be fed into local search procedure to obtain further improvement.