Machine Scheduling

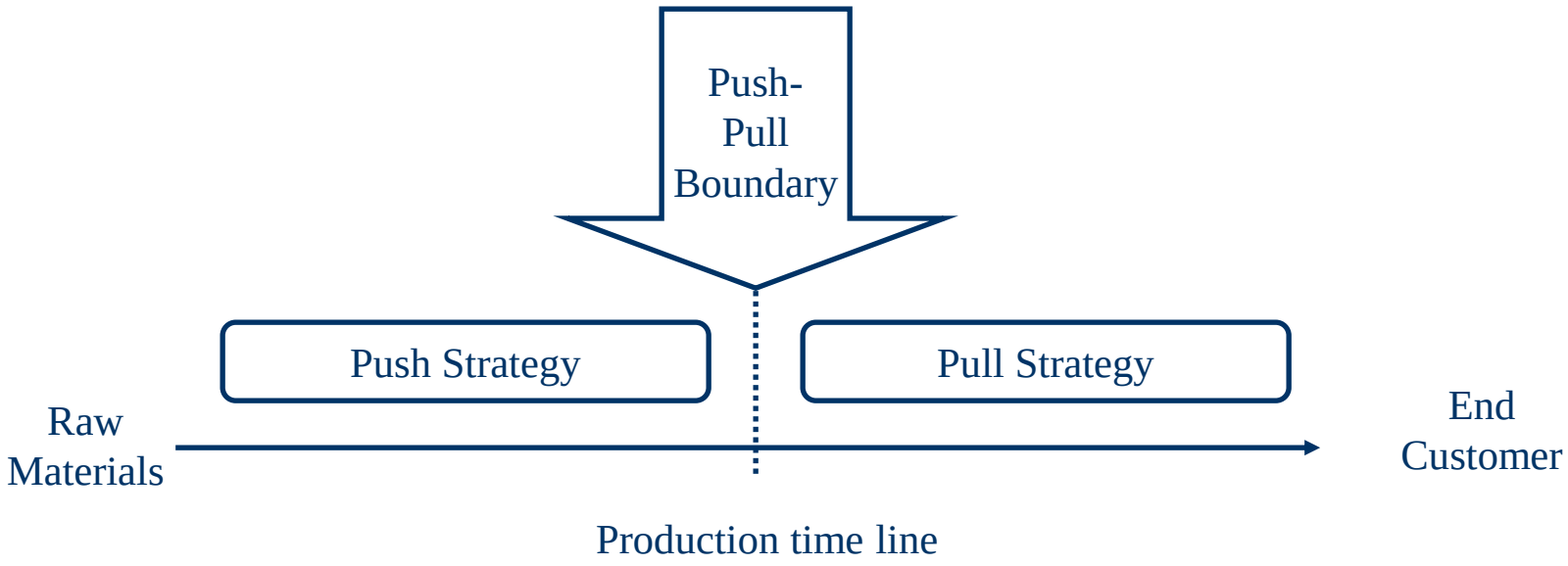
IE375 Spring 2020 On-Line

Nesim K. Erkip

Machine Scheduling

- A ***schedule*** is a tangible plan that tells us when certain activities will happen. Schedule often tells us the *sequence* of activities.
- ***Scheduling*** is the process of generating a schedule.
- ***Machine scheduling*** is generating the schedule of activities that needs to take place in the shop floor. It is often called *shop floor control*.
- Machine scheduling is usually the lowest hierarchical level of decision making in an organization. It needs input from several upstream planning decisions.

Push-Pull Systems (Plan)



Hierarchy of Planning Problems

Definitions – (jargon)

- A ***machine*** is a resource that can perform at most one activity at any time.
- Activities are commonly referred to as ***jobs***, and it is assumed that a job is worked on by at most one machine at any time.
- Jobs are processed on machines for a time period that is called ***processing time***.
- In general, a scheduling problem is one in which ***n* jobs** must be processed through ***m* machines**.
- There may be many different ***optimization crite*ria** and constraints on ***job sequences*** that may impact the complexity of the problem

Definitions

- *Processing time* of job *j* on machine *i* (t_{ij})
- *Start time* of job *j* on machine *i* is the time that machine *i* starts processing job *j*.
- *Completion time* of job *j* on machine *i* is the time that machine *i* finished processing job *j*. If there is no interruption,

  completion time = start time + processing time

- Completion time (C_j) of a job in the job shop is the time that the job is completed in all machines required for the processing of that job. Completion time of a job is the maximum of completion of that job across all machines.
- *Due date (d_j)* is the time that a job is required (or expected) to be completed in all machines that it is required of.

Broad objectives

- *Turnaround* measures the time required to complete a task.
- *Timeliness* measures the conformance of a particular task's completion to a given deadline.
- *Throughput* measures the amount of work completed during a fixed period of time.

More definitions

- *Ready time* (r_j) is the time at which the job is ready (or available) for processing
- *Flow time* (F_j) is the time that the job spends in the system
- *Lateness* (L_j) is the difference between the completion time and the due date ($L_j=C_j-d_j$).
- *Tardiness* is the positive difference between the completion time and the due date of a job

  ($T_j=max\{C_j-d_j,0\}$). A tardy job is one that is completed after its due date.
- *Makespan* is the time that all jobs are completed in the job shop. $Makespan=max\{C_j\}$

Example

- A machining center in a job shop for a local fabrication company has five unprocessed jobs remaining at a particular point in time. The jobs are labeled 1, 2, 3, 4, and 5 in order that they entered the shop. The processing times and due dates are as follows

Job number	Processing Time	Due Date
1	11	61
2	29	45
3	31	31
4	1	33
5	2	32

Objectives - Example

- An air traffic controller is faced with the problem of scheduling the landing of five aircrafts. Based on the position and runway requirements of each plane, he estimates the following landing times:

Plane	1	2	3	4	5
Time (in minutes)	26	11	19	16	23

- The planes are of different sizes

Plane	1	2	3	4	5
Number of passengers	180	12	45	75	252

- Each flight has a scheduled arrival time

Plane	1	2	3	4	5
Scheduled arrival time	5:30	5:45	5:15	6:00	5:40

- Plane number 4 has a critically low fuel level

Some objectives

- Minimize
  - Makespan
  - Total flow time; Weighted flow time; Maximum flow time
  - Total tardiness; Weighted tardiness; Maximum tardiness
  - Total lateness; Total earliness
  - Number of tardy jobs
- Typically each of the performance measures is a function of job completion times $Z=f(C_1,C_2,..,C_n)$. An objective is a ***regular* objectiv**e (or measure) if
  - If the objective is to minimize Z
  - If f is a non-decreasing function of C_j

Specific sequencing rules

- FCFS (First-come, first-served). Jobs are processes in the sequence in which they enter the shop
- SPT (shortest processing time). Jobs are sequenced in increasing order of their processing times
- EDD (earliest due date). Jobs are sequenced in increasing order of their due dates
- CR (critical ratio). Dynamically sequence jobs based on: (Due date-current time)/processing time
- Any rule

Example

Job number	Processing Time	Due Date
1	11	61
2	29	45
3	31	31
4	1	33
5	2	32

- Plan using 4 different sequencing rules

Different issues in scheduling

- Job arrival pattern
- Preemptive versus non-preemptive
- Setup times dependence on job sequence
- Deterministic versus stochastic processing times
- Availability of machines
- Monolithic jobs versus lot streaming

Single machine scheduling

- Basic assumptions
  - Deterministic processing times
  - Machine available at all times
  - No preemption
  - No setup times (or setup times part of processing times)
  - All jobs are ready at time 0.

Single machine scheduling

- Every schedule gives the same makespan for the single machine scheduling problem.
- *Theorem:* Mean flow time is minimized by Shortest Processing Time (SPT) scheduling rule
  - Mean Flow Time is a good measure for inventory in a system
- *Theorem:* Total weighted flow time is minimized by Shortest Weighted Processing Time (SWPT) scheduling rule

Problems with due dates

- *Theorem*: Total lateness is minimized by SPT sequencing rule

- *Theorem*: Maximum lateness and maximum tardiness are minimized by Earliest Due Date (EDD) sequencing rule

Difficult problems with due dates

- Minimizing weighted number of tardy jobs is NP-hard
- Minimizing total tardiness is NP-hard.
- Minimizing total weighted tardiness is NP-hard.

- Difficult problems – NP-hard – see the slides at the end.

Minimizing the number of tardy jobs (Moore's Algorithm)

- Step 1: Sequence the jobs with EDD
- Step 2: Find the first tardy job in the current sequence, say job [i]. If none exists go to step 4.
- Step 3: Consider the jobs [1],[2],…,[i]. Reject the job with the largest processing time. Return to step 2.
- Step 3: Form an optimal sequence by taking the current sequence and appending to it the rejected jobs. The jobs appended to the current sequence may be in any order.

Example

- A machine shop processes custom orders from a variety of clients. One of the machines, a grinder, has six jobs remaining to be processed. The processing times and due dates are given below. What is the sequence that minimizes the number of tardy jobs

Job	1	2	3	4	5	6
Due Date	15	6	9	23	20	30
Processing time	10	3	4	8	10	6

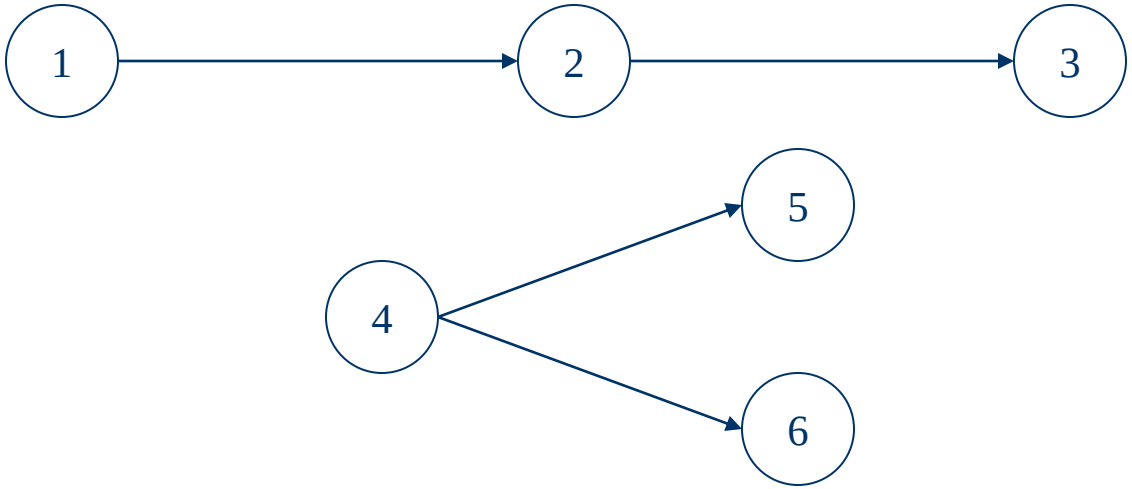
Single machine problems with precedence constraints – Lawler's algorithm

- Consider any objective where we are minimizing a maximum of $g_i(F_i)$ where g_i is any non decreasing function of flow times
- In addition, there are precedence constraints in that certain jobs must be completed before other jobs can begin
- Lawler's algorithm first schedules the job to be completed last, then the job to be completed next to last, and so on.
- At each stage, one determines the set of jobs not required to precede any other. Among this set, pick the one that gives the minimum of maximum of $g_i(F_i)$

Example

- Consider the following jobs with due dates and precedence constraints

Job	1	2	3	4	5	6
Processing time	2	3	4	3	2	1
Due date	3	6	9	7	11	7



Multiple Machines- Parallel machines

- There are *m* machines that are doing exactly the same thing.
- Each job can be processed in any of the *m* parallel machines.
- If we allow for pre-emption, minimum makespan is

  $M^* = max[\Sigma p_j/m, max_j\{p_j\}]$

- If we do not allow for pre-emption, finding the minimum makespan is an NP-hard problem
- A ***list schedule*** picks up a job from a list and places it to a machine which is available at the earliest time.
- Any list schedule gives a makespan *M* such that

  $M/M^* \leq 2 - 1/m$

- An LPT list schedule gives a makespan *M* such that

  $M/M^* \leq 4/3-1/3m$

Parallel machines- Examples

- With pre-emption, 3 machines

  $p_1=1$ $p_2=2$ $p_3=3$ $p_4=4$ $p_5=5$ $p_6=6$ $p_7=8$ $p_8=8$

- Without pre-emption, 4 machines

  $p_1=3$ $p_2=3$ $p_3=3$ $p_4=1$ $p_5=1$ $p_6=1$ $p_7=4$

Multiple machines – Series machines

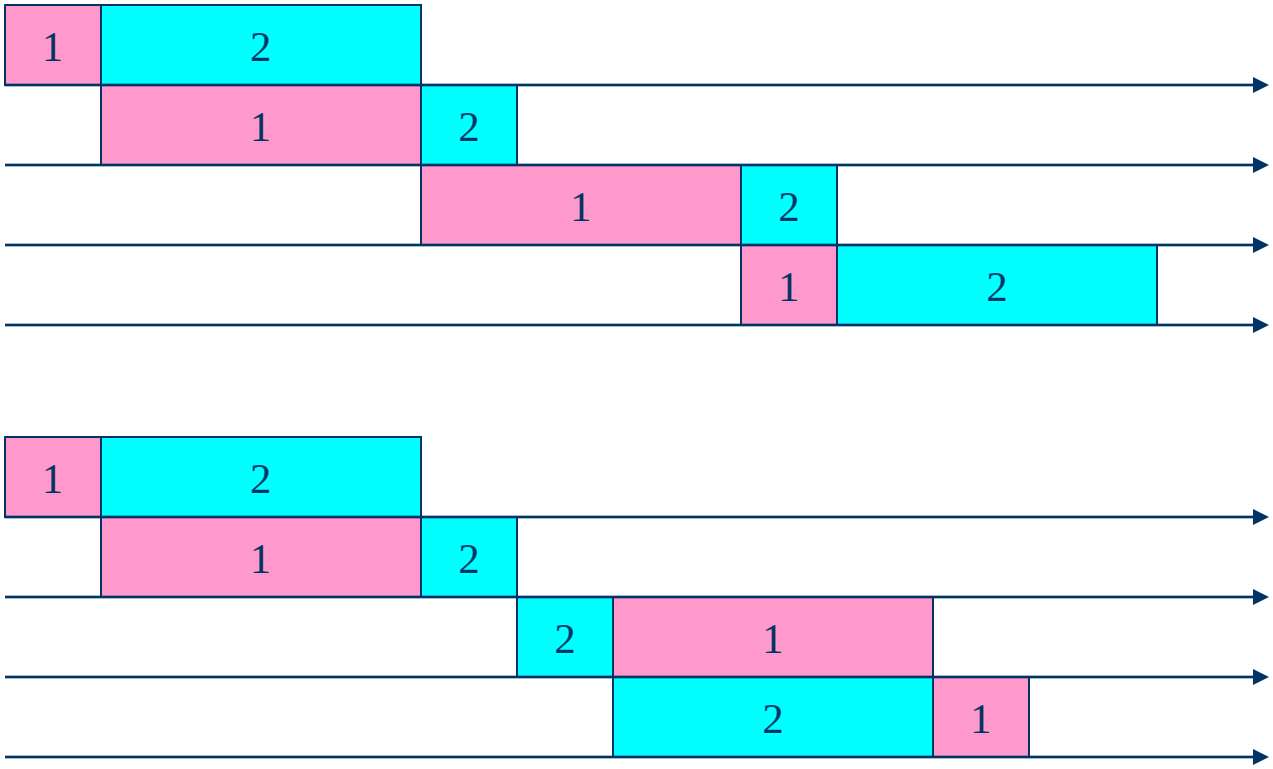
- Each job needs to be processed on different machines in “some” order
- If the order is same for all jobs, then the shop is called a ***flow shop***
- If the order is fixed a-priori, and can be different for each job, then the shop is called a ***job shop***
- If the order is not fixed a-priori, and can be different for each job, then the shop is called an ***open shop***

Flow shops

- Each job needs to be processed in machines *1,2,3,..,m* in that order
- p_{ij} is the processing time of job *j* on machine *i*
- Consider two jobs to be processes in 4 machines

Job	1	2
p_{1j}	1	4
p_{2j}	4	1
p_{3j}	4	1
p_{4j}	1	4

Flow shop general example



Flow shops

- Permutation schedules are optimal for any regular measure if there are 2 machines
- Permutation schedules are optimal for minimizing makespan if there are 3 machines
- Minimizing makespan in two machine flow shop (Johnson's rule)
  - Step 0:
    - A_j=Processing time of job j on machine A
    - B_j=Processing time of job j on machine B
  - Step 1: List the values of A_j and B_j in two columns
  - Step 2: Find the smallest remaining element in two columns. If it appears in Column A, then schedule that job next. If it appears in Column B, the schedule that job last.
  - Step 3: Cross of the jobs as they are scheduled. Stop when all jobs have been scheduled.

Two machine flow shop example

- Minimize the makespan for the following flow shop.

Job	Machine A	Machine B
1	5	2
2	1	6
3	9	7
4	3	8
5	10	4

Three machine flow shop

- Minimizing makespan in the general 3 machine flow shop is NP-hard.
- If min $A_i \geq \max B_i$ or min $C_i \geq \max B_i$, then, solve a two machine problem with $A'_i = A_i + B_i$ and $B'_i = B_i + C_i$
- Solve th e following three machine problem to minimize makespan

Job	Machine A	Machine B	Machine C
1	4	5	8
2	9	6	10
3	8	2	6
4	6	3	7
5	5	4	11

Scheduling Algorithms and Complexity

- How do we know that an algorithm to solve a scheduling problem is a “good” algorithm?
- One important measure of performance is the “rate of growth of the time or space required to solve larger and larger instances of a problem”.
- The size of the problem is measured by the size of the input data.
- For example, for a one machine scheduling problem, the number of jobs, n, is the size of the problem.
- The *time complexity* of the problem is the time needed by the algorithm (e.g., number of elementary operations such as additions or comparisons) expressed as a function of the problem size.
- Similarly for *space complexity*.

Polynomial algorithms

- If a scheduling problem solves, in the worst case, a problem with an input size of n in time cn^2 for some constant c, then we say the time complexity of the algorithm is $O(n^2)$.
- Generally $O(p(n))$, where $p(n)$ is a polynomial function.
- A polynomially bounded algorithm is a “good” algorithm and the problem is “well-solved”.
- Consider two scheduling algorithms for the same problem

Algorithm	Time Complexity	Maximum problem size		
		1 sec	1 min	1 hour
A_1	n log n	140	4893	2.0×10^5
A_2	2^n	9	15	21

Optimization versus feasibility

Optimization Problem

$$\underset{x \in X}{Min}\ f(x)$$

Feasibility problem

Is there any $x \in X$ such that $f(x) \leq z$?

- An optimization problem can be solved by solving polynomial number of corresponding feasibility problems
- Therefore for complexity issues it is sufficient to consider the feasibility problems.
- The *feasibility* problems are also called *recognition, verification* or *decision* problems

Classes P and NP

- *Class P*: A feasibility problem belongs to class *P,* if for any instance of the problem, a “yes” or a “no” answer can be *determined* in polynomial time
- *Class NP*: A feasibility problem belongs to class *NP*, if the feasibility of a given structure can be *checked* in polynomial time.
- Example:
  - Is there any sequence of jobs whose total tardiness is less than x?
  - Does a given schedule have total tardiness less than x?
- It is known that *P* is equal to or a subset of *NP*.
- It is still debatable whether *P=NP*. If *P=NP*, all problems in *NP* can be solved in polynomial time.
- An *NP-complete* problem is, roughly speaking, a hardest problem in *NP*, in that if it would be solved in polynomial time, then each problem in *NP* would be solved in polynomial time, so that *P=NP.*
- NP-completeness of a particular problem is a strong evidence that a polynomial time algorithm is unlikely to exist.
- A problem p can be shown to be NP-complete if a known NP-complete problem can be reduced to problem p in polynomial time.

Some well-known NP-complete problems

- Set partitioning problem

Given numbers $a_1, a_2, a_3, ..., a_n$ and $\sum_{i=1}^{n} a_i = 2b$

Is there a subset $S \subset \{1,2,3,..,n\}$ such that $\sum_{i \in S} a_i = b$?

- Knapsack problem

Given numbers $a_1, a_2, a_3, ..., a_n$, b

Is there a subset $S \subset \{1,2,3,..,n\}$ such that $\sum_{i \in S} a_i = b$?

- Traveling salesman problem

NP-hard problems

- Theorem: Minimizing weighted number of tardy jobs in a single machine shop is NP-Hard

- Theorem: Minimizing total tardiness in a single machine shop is NP-Hard

What next?

- Find instances where easier solutions exists
  - Two jobs are *agreeable* if $p_i >= p_j$ implies that $d_i >= d_j$
  - For any pair of jobs, assume that processing times and due dates are *agreeable*. Then total tardiness is minimized by SPT or equivalently EDD.
- Find dominant schedules
  - If jobs *i* and *j* are the candidates to begin at time *t*, then the job with the earlier due date should come first, except if

    $$t+max\{p_i,p_j\}>max\{d_i,d_j\}$$

    in which case the shorter job should come first if the objective is to minimize total tardiness

What next?

- Exact methods
  - Complete enumeration (n! complexity)
  - Implicit enumeration
    - Dynamic programming methods ($n2^n$ complexity): start with smaller problems and dynamically solve larger problems building on the solution of smaller problems
    - Branch and bound methods: partition a large problem into two or more sub-problems, calculate a lower bound on the optimal solution of a given problem and fathom the sub-problem if there is a solution elsewhere with value below the lower bound
  - Use dominance rules, and special cases to reduce the complexity of the dynamic programming formulation or branch & bound method

Heuristic methods

- Dispatching and construction procedures
  - Dispatching: find the next job to be scheduled using “some” rule
  - Construction: build a schedule adding jobs to the scheduled one at a time, but not necessarily from earliest to the latest
  - Insertion: keep the relative order of existing jobs fixed, insert the next job in the best place
  - Sometimes called “greedy” procedure since it makes the selection in most favorable way, without regard to the possibilities that might arise later

Heuristic methods

- Neighborhood search techniques
  - *Step 1:* Obtain a sequence to be an initial seed and evaluate it with respect to the performance measures
  - *Step 2:* Generate and evaluate all the sequences in the neighborhood of the seed. If none of the sequences is better than the seed with respect to the performance measure, stop. Otherwise proceed.
  - *Step 3:* Select one of the sequences in the neighborhood that improved the performance measure. Let this sequence be the new seed. Return to *Step 2*.
- Tabu search
  - Instead of stopping when a local optimum is encountered, tabu search accepts a new seed, even if the value is worse than that of the current seed.
  - If the new seed is worse than the previous seed, the procedure could cycle indefinitely. Therefore a move back to the previous seed is designated as tabu.
  - Procedure stops when a predetermined number of moves are performed

Heuristic Methods

- Simulated annealing
  - Similar to neighborhood search
  - Always move to a neighborhood solution if it is better than the current seed.
  - If the new solution is worse than the seed, there is still some chance that the procedure will move to the next solution but with that chance reducing over time and inversely related with the “worseness”
- Genetic algorithms
  - Mix pieces of different good schedules to obtain a better solution