Week 1

*If the number of odd degrees is 0 or 2, the connected path has an Euler path.*
*If 0, the path is a cycle: starts and finish at the same vertex.*

*Shortest path Dijkstra:*
*Look at each vertex, which is path shorter, and compare, choose the shortest path.*

*A spanning tree:*
*A spanning tree is a sub graph, which contains all the vertices and is a tree. A tree is a connected graph without any cycles.*
*A graph may have more spanning trees.*

*Minimal spanning tree:*
*The spanning tree with the min cost for that graph*

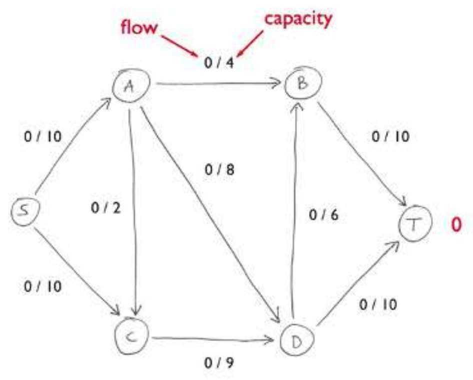
- ➔ *Kruskal's and Prim's algorithm for minimal spanning tree*

*Kruskal's algorithm:*
1) *Sort all the edges in decreasing order*
2) *Connect the edges in this order*
3) *If there's a cycle, don't add the edge*

*Prim's algorithm:*
1) *Start random*
2) *Choose the shortest path from your starting point*
3) *Look which path is the shortest from the vertices you already reached*

*Augmenting path has:*
1) *Non-full forward edges*
2) *Non-empty backward edges*

*Max flow: Ford Fulkerson algorithm:*
*For computing the <u>max flow</u> in flow network*
*From source to sink*
1) *Find an augmenting path*
2) *Compute the bottleneck capacity*
3) *Augment each edge and the total flow*




*MaxFlow=MinCut*
*The running time only depends on the size of the network and not on the capacities.*

*Max flow: Edmonds-Karp-Dinitz (EKD) algorithm:*

*This algorithm applies the FF algorithm, but in each iteration it choose the s, t path in the residual graph with the minimum number of arcs.*

*Min flow: The cycle cancelling algorithm:*

1) *Find a feasible flow of value v. Make the residual graph*
2) *While there is negative cost cycle C in the residual graph:*

➔ *Add the largest possible flow over C*

➔ *Update the residual graph*

*What kind of questions to expect?*

- *Give the corresponding LP-formulation for the network*

➔ *Make all the paths*

➔ *Number them X1, X2, ..., Xn -> max X1, X2..*

➔ *Number all the edges*

➔ *Look at which path crosses the edge, and what the capacity is*

➔ *You get St. X1 <= capacity passing edge*

- *Give the maximum flow and it's corresponding LP-solution*

➔ *Maximum flow: look at all the edges capacity, add al the capacities of the paths*

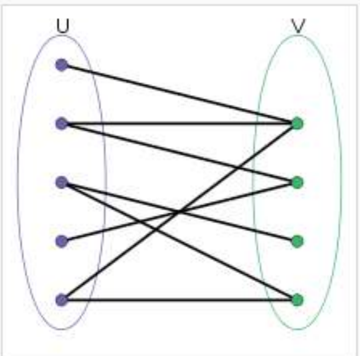
➔ *Max flow: what passes all edges*

- *Give the dual of this LP*

➔ *Capacity times the path becomes -> min 2Y2 + 3Y2*

➔ *St. path -> Y1 + Y3 >= 1*

- *Find a solution to the dual with value equal to the primal solution found*
- *Give an interpretation of this dual solution in terms of the network*
- *Find a minimum cost flow by using the cycle cancelling algorithm.*
- *Bipartite cycles:*




Example of a bipartite graph without cycles

*An arc a is called upward critical if increasing the capacity of a increases the value of the maximum flow*
*An arc a is called downwards critical if decreasing the capacity of a decreases the value of the maximum flow*

## *Week 2*

*P vs. NP:*

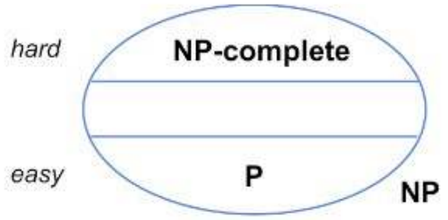
*A problem is in NP hard if it can be verified in polynomial time.*
*A problem is in P if a solution can either be found or proven in polynomial time.*

*P is a subset of NP. Not really correct!*

*A search problem is called NP-complete if all other search problems reduce to it.*
*A problem is called NP-hard if all other search problems reduce to it.*
*(Notice the difference, search problem and problem)*

*NP-complete problems can be seen as the hardest problems among all the search problems.*

*The halting problem is an example of an NP-hard problem that is not in NP.*




*Reductions:*

The implication of $A \to B$ is twofold:

1. Any efficient algorithm for $B$ can be used to solve $A$ efficiently.
2. Solving problem $B$ is at least as difficult as solving problem $A$ (up to a polynomial factor in running time).

Reductions are transitive (they compose):

$$(A \to B \text{ and } B \to C) \quad \Rightarrow \quad A \to C.$$

*The class of search problem is also known as NP.*

*Every optimization problem can be modelled as a search problem.*
*Any algorithm for search problems can be used for optimization problems, with just a polynomial factor loss in running time.*
*If we have an algorithm for the search problem, we also have an algorithm for the decision version.*

*If A -> B then any algorithm for B can be used to solve A.*
*If A -> B and A is NP-complete, then B is NP-complete as well.*


Figure 2: Three forms of optimization. Some search problems do not have an optimization variant.

| **P** (easy) | **NP**-complete (hard) |
|---|---|
| Euler tour | Rudrata tour |
| Shortest path | Longest path |
| Chinese Postman | Traveling Salesman |
| Linear Programming (LP) | Integer Linear Programming (ILP) |
| 2SAT | Satisfiability (SAT) and 3SAT |
| Minimum Spanning Tree | Vehicle Routing |
| Matching | Load Balancing |
| Maximum Flow | Knapsack |
| Minimum Cost Flow | Vertex Cover |
| ... | ... |

Figure 4: Although both lists are essentially infinitely long, most problems in practice are hard. Some problems come in easy-hard pairs like Euler-Hudrata and LP-ILP.

*Important:*
*f=O(g) -> g goes up, f goes down*
*f= Ω(g) -> g goes down, f goes up*
*f=Θ(g) -> g and f goes same way*

## *Week 3*

*See lecture nodes:*

- *Approximation algorithms*
- *Vertex cover*
    - *Algorithm A: maximal matching*
    - *Algorithm B: LP-rounding*
    - *Generalization to Set Cover*
- *K-Clustering: Greedy algorithm*
- *The traveling salesman (TSP)*
    - *Complexity of TSP*
    - *Double tree algorithm*
    - *Nearest addition algorithm*
    - *Christofides' algorithm*

## Approximation Algorithms

**Definition 1.** *An $\alpha$-approximation algorithm for an optimization problem is a polynomial-time algorithm that, for each instance of the problem, produces a solution with a value that is within a factor $\alpha$ of the optimal value.*

For an instance $I$, we deonte by $\textsc{Opt}(I)$ the optimal value and by $\textsc{Alg}(I)$ the value returned by the algorithm.

To show that an algorithm is an $\alpha$-approximation algorithm we need to show three things:

(1) The algorithm runs in polynomial time.

(2) The algorithm always produces a feasible solution.

(3) For any isntance $I$, the value is within a factor $\alpha$ of the optimal value:
$\textsc{Alg}(I) \leq \alpha\textsc{Opt}(I)$ (for a minimization problem, $\alpha \geq 1$)
$\textsc{Alg}(I) \geq \alpha\textsc{Opt}(I)$ (for a maximization problem, $\alpha \leq 1$)

*Approximation algorithms deal with optimization problems. The algorithm should always return a solution with a value that is close to the optimal value.*

## 1. Vertex cover

In this problem, we need to find for a given graph $G = (V, E)$ a subset of vertices such that each edge has an endpoint in the set. The goal is to minimize the number of vertices in the subset.

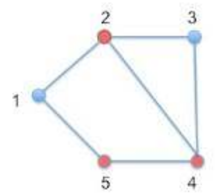



Figure 1: *The red vertices form a minimum vertex cover:* $S = \{2, 4, 5\}$.

VERTEX COVER:

| | |
|---|---|
| *Instance*: | Graph $G = (V, E)$. |
| *Output*: | $S \subseteq V$ such that each edge has at least one endpoint in $S$. |
| *Goal*: | Minimize $\lvert S \rvert$. |

*It is a NP-hard problem. Thus there is no polynomial time algorithm that solves the problem, unless P=NP.*

**Algorithm A: maximal matching**
*Find a maximal matching M and add all endpoints of the edges in M to S.*
*(Algorithm A is a 2-approximation algorithm)*

**Algorithm B: LP-rounding**
*LP's can be solved in polynomial time*

The vertex cover problem can easily be formulated as an integer linear programming problem (ILP). Let $n = |V|$ be the number of vertices.

$$\begin{aligned} \text{(ILP)}\ \min \quad & Z = \sum_{j=1}^{n} x_j \\ s.t. \quad & x_i + x_j \geqslant 1 \quad \text{for all } (i,j) \in E \\ & x_j \in \{0,1\} \quad \text{for all } j \in V. \end{aligned}$$

The vertex cover problem is $\mathcal{NP}$-hard which implies that the ILP above can not be solved in polynomial time, unless $\mathcal{P}=\mathcal{NP}$. However, the following LP-relaxation (in which $x_j \in \{0,1\}$ is replaced by $x_j \geq 0$) can be solved efficiently.

$$\begin{aligned} \text{(LP)}\ \min \quad & Z = \sum_{j=1}^{n} x_j \\ s.t. \quad & x_i + x_j \geqslant 1 \quad \text{for all } (i,j) \in E \\ & x_j \geqslant 0 \quad \text{for all } j \in V. \end{aligned}$$

The idea of the algorithm is to solve (LP) and then *round* that solution in a feasible solution for (IP). This technique is called *LP-rounding*.

*(Instead of Xj in {0,1} -> Xj >= 0)*

## Weighted Vertex Cover problem

*In this case each vertex has a given weight Wj>0 and the goal is to minimize the total weight of the cover.*

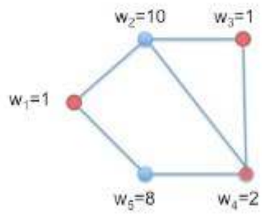



Figure 4: *Graph G with weights on the vertices is an instance of the weighted Vertex Cover problem. The optimal solution has total weight* $1+1+2=4$.

$$\begin{aligned} \text{(LP)}\ \min \quad & Z = \sum_{j=1}^{n} w_j x_j \\ s.t. \quad & x_i + x_j \geqslant 1 \quad \text{for all } (i,j) \in E \\ & x_j \geqslant 0 \quad \text{for all } j \in V. \end{aligned}$$

[3] The value of the solution found is

$$\sum_{j\in S} w_j = \sum_{j=1}^{n} w_j \hat{x}_j \leqslant 2 \sum_{j=1}^{n} w_j x_j^* = 2Z_{LP}^* \leqslant 2Z_{ILP}^* = 2\text{OPT}.$$

**Generalization to Set Cover**

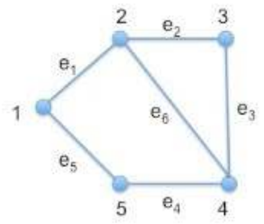



Figure 5: *Graph G is an instance of the Vertex Cover problem. Equivalently, we can write it as a Set Cover problem. For each vertex j there is a set $S_j$ containing the adjacent edges: $S_1 = \{e_1, e_5\}$, $S_2 = \{e_1, e_2, e_6\}$, $S_3 = \{e_2, e_3\}$, $S_4 = \{e_3, e_4, e_6\}$, and $S_5 = \{e_4, e_5\}$.*

and assume that each item appears in at most $f$ sets, for some constant $f$. The LP-rounding algorithm for vertex cover problem applies here in the same way.

$$\text{(ILP)} \min \quad Z = \sum_{j=1}^{n} w_j x_j$$
$$s.t. \quad \sum_{j:e_i \in S_j} x_j \geqslant 1 \quad \text{for all } i = 1, \ldots, m$$
$$x_j \in \{0, 1\} \quad \text{for all } j = 1, \ldots, n.$$

The LP-relaxation is obtained by replacing $x_j \in \{0, 1\}$ by $x_j \geq 0$.

$$\text{(LP)} \min \quad Z = \sum_{j=1}^{n} w_j x_j$$
$$s.t. \quad \sum_{j:e_i \in S_j} x_j \geqslant 1 \quad \text{for all } i = 1, \ldots, m$$
$$x_j \geqslant 0 \quad \text{for all } j = 1, \ldots, n.$$

**Algorithm** $\mathcal{B}$ (set cover):
Step 1: Solve the LP. $\rightarrow$ Optimal values $x_1^*, x_2^*, \ldots, x_n^*, Z_{LP}^*$
Step 2: Let $U$ be all $j$ for which $x_j^* \geqslant 1/f$.

*What kind of questions to expect?*

- *Show with an example that algorithm A is a 2-approximation algorithm for the weighted vertex cover problem*
  **If true:** *ALG/OPT <= 2*
  **If not true:** *ALG/OPT >2*
- *Give an optimal vertex cover for the graph*
- *Give the ILP for this vertex cover*
  *Min X1 + X2 +*
  *St. X1 + X2 >= 1*
  *Xi in {0,1} for i=1, 2,*
- *Give the LP relaxation for this vertex cover*
  *Min X1 + X2 +*
  *St. X1 + X2 >= 1*
  *Xi >= 0 for i=1, 2,*
- *Give a solution to the LP-relaxation which is strictly smaller/bigger than the optimal value*

## 2. The $k$-cluster problem.

K-CLUSTER:

*Instance*: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance <u>metric</u> $d(,)$ and an integer $k$.

*Output*: A partition of the points into $k$ clusters $C_1, \ldots, C_k$.

*Goal*: Minimize the maximum diameter of a clusters:

$$\text{Minimize:} \quad \max_j \left\{ \max_{x_a, x_b \in C_j} d(x_a, x_b). \right\}$$

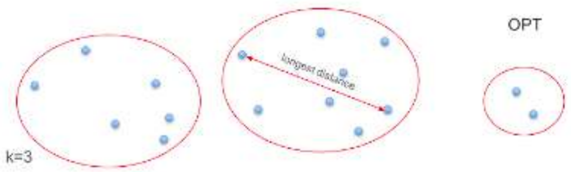



Figure 6: Example. The cost of the solution is the maximum distance betwee two points in a cluster.

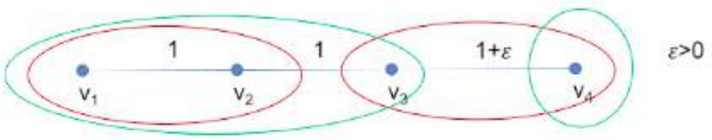
**Algorithm** Greedy:

- Pick the first center $\mu_1$ arbitrarily.
- For $i = 2$ to $k$:
  Let $\mu_i$ be the point in $X$ that is farthest from $\{\mu_1, \ldots, \mu_{i-1}\}$.
- Create $k$ clusters: $C_i$ is the set of all $x \in X$ whose closest center is $\mu_i$.

**Theorem 4.** *The Greedy algorithm is a 2-approximation algorithm.*

**Exercise 3** Show by an example that the Greedy algorithm for $k$-clustering is not better than a 2-approximation algorithm. That means, given an example for which the value of the algorithm's solution is twice the optimal value.

*Solution*: Take for example 4 points on a line with distances as shown and with $k = 2$. The optimal solution has maximum diameter equal to 1. If the greedy algorithm starts with point $v_1$, then $v_4$ will be the other center. The clusters are $\{v_1, v_2.v_3\}$ and $\{v_4\}$ and the maximum diameter is 2.
The ratio ALG$/OPT \to 2$ for $\epsilon \to 0$.




**The traveling salesman (TSP)**

*A complete graph with a cost Cij for every pair i, j. A cycle that goes trough every vertex exactly once. The goal is to minimize the length of the cycle.*

*There are three algorithms for TSP:*

**Algorithm** 1 (Double tree).

- Find a minimum spanning tree $T$.
- Double all the edge of the tree. (See Figure 10).
- Find an Euler tour in the double tree.
- Apply shortcutting in order to turn the Euler tour into a Hamiltonian cycle.

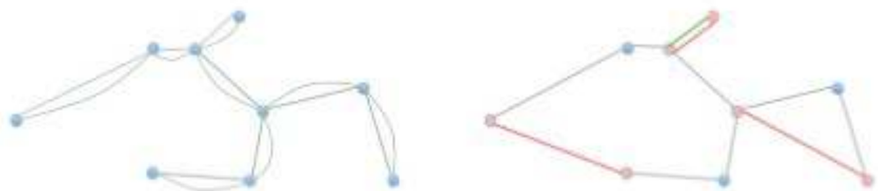


Figure 10: *The double tree (left) (for Algorithm 1) and the MST plus a matching of the odd-degree nodes (right) (for Algorithm 3).*

**Algorithm** 2 (Nearest addition).

- Pick an arbitrary point, say $i_1$, as the first point.
- Let $i_2$ be the point nearest to $i_1$. Make a directed tour from $i_1$ to $i_2$ and back to $i_1$. Let $S = \{i_1, i_2\}$.
- Repeat the following until a feasible tour is found:
  - Find a pair $i \in S, j \notin S$ with minimum cost $c_{ij}$. (In other words, find the point $j$ that is nearest to the already chosen set $S$.) Insert $j$ in the tour after $i$. Add $j$ to $S$.

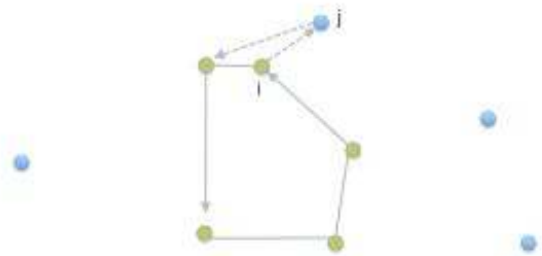



Figure 9: *Iteration of the nearest addition algorithm.*

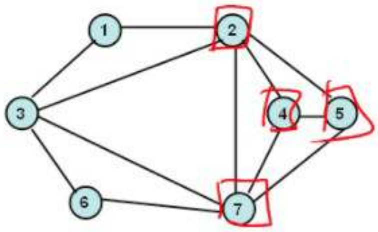
**Algorithm** 3 (Christofides' algorithm).

- Find a minimum spanning tree $T$. Let $O$ be the vertices of odd degree in $T$.
- Find a minimum cost perfect matching of the vertices in $O$. Denote the edges in this matching by $M$.
- Find an Euler tour in the graph $T + M$.
- Apply shortcutting in order to turn the Euler tour into a Hamiltonian cycle.

Note that a perfect matching on $O$ exists since $|O|$ is even. (Any graph contains an even number of odd-degree points.) Also note that the cheapest perfect matching can be found in polynomial time. (Not for this course.)

# Clique

- Clique
  - Graph G = (V, E), a subset S of the vertices is a clique if there is an edge between every pair of vertices in S

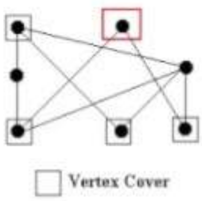



## Vertex Cover

➢ Definition:

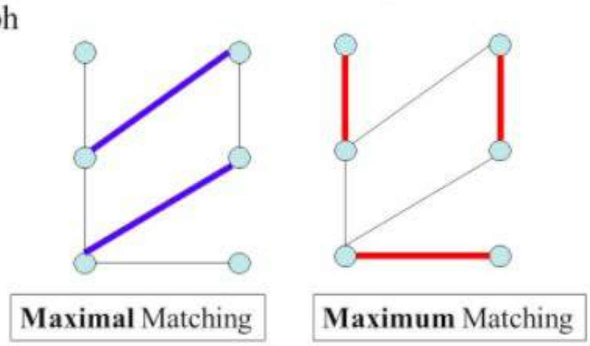
Given an undirected graph $G = (V, E)$, a subset $V' \subseteq V$ is called a vertex cover of $G$ iff for every edge $e \in E$, $e$ has at least one endpoint incident at $V'$

➢ An Example




# Maximal & Maximum Matchings

- **Maximal Matching:** A maximal matching in a graph is a matching that cannot enlarged by adding an edge
- **Maximum Matching:** A maximum matching is a matching of maximum size among all matchings in the graph




*Graph isomorphism:*
*If there is isomorphism then there's bijection between the vertex set of two graphs.*

*Rudrata's path:*
*Visit all vertices of a graph exactly once.*

*Rudrata's cycle:*
*Visit all vertices of a graph exactly once and end at the starting point*

*Hamilton cycle:*
*Visit all vertices of a graph exactly once and end at the starting point*

*Rudrata's cycle = Hamilton cycle*

*Euler path:*
*Visit all edges of a graph exactly once.*

*Big 0 notation*

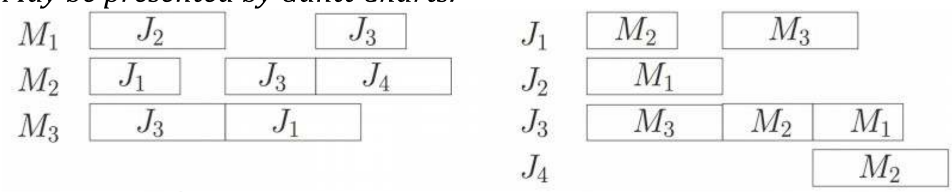
## *Week 4*

*Scheduling*

*What is scheduling?*
*Scheduling concerns optimal allocation or assignment of resources, over time, to a set of tasks/ activities/ jobs.*

*Resources (M): machines, people, space*
*Tasks (J): production, jobs, classes, flights*

*May be presented by Gantt Charts:*




- m machines   i=1,…,m
- n jobs   j=1,…,n

**Job parameters:**

- $p_j$ : processing time of job j
- $p_{ij}$ : processing time of job j on machine i
  (when processing time of job j depends on machine i)
- $r_j$ : release date of job j   (earliest starting time)
- $d_j$ : due date (deadline)   (=committed completion time)
- $w_j$ : weight of job j   (importance)

# Classification of Scheduling Problems

(Most) scheduling problems can be described by a three field notation $\alpha|\beta|\gamma$, where

- $\alpha$ describes the <u>machine</u> environment
- $\beta$ describes the <u>job</u> characteristics, and
- $\gamma$ describes the <u>objective</u> criterion to be minimized (or max.)

Remark: A field may contain more than one entry but may also be empty

**Example:**

$1 \mid r_j \mid \Sigma_j C_j$ — Single machine. Jobs have release times. Objective is minimizing the sum of the completion times.


## Machine environment ($\alpha$)

- **Single** machine ($\alpha = 1$)

*Why single machines?*

- *The*

- **Identical parallel** machines ($\alpha$ = P or Pm)
  - m identical machines running in parrallel;
    - If $\alpha$ = P, then the number of machines, m, is part of the input
    - If $\alpha$ = Pm, the value m is considered a constant
  - each job consist of a single operation and this may be processed by any of the machines for $p_j$ time units

- **Unrelated parallel** machines ($\alpha$ = R or Rm)
  - m different machines in parallel
    $p_{ij}$ is the process time of job j if scheduled completely on machine i

*simply arrive in practice*

- *Multi machine problems can often be decomposed into single machine problems*
- *The form the basic for the design of the algorithms for more complicated scheduling problems*

## Job characteristics (β )

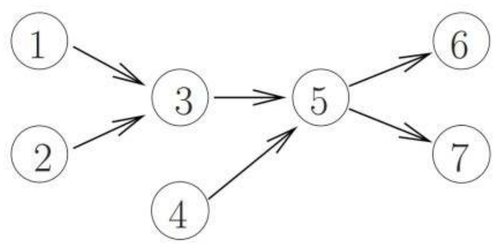
- release dates ($r_j$ in β field)
  - if $r_j$ in β field, jobs may not start processing before their release date
  - if $r_j$ is not in β field, jobs may start at any time

- deadlines ($d_j$ in β field)
  - if $d_j$ is in β field, each job j should finish before time $d_j$

- preemption (*pmtn* in β field)
  - processing of a job on a machine may be interrupted and resumed at a later time even on a different machine

- unit processing times ($p_j = 1$ or $p_{ij} = 1$ in β field)
  - each job (operation) has unit processing times

- precedence constraints (prec in β field)
  - A job cannot start before some other job(s) are finished
  - May be represented by an acyclic graph (vertices = jobs, arcs = precedence relations)




For example: job 5 can not start before 1,2, 3 and 4 are completed.
jobs 1,2, and 4 can start immediately.

## Objective function (γ)

**Notation:**

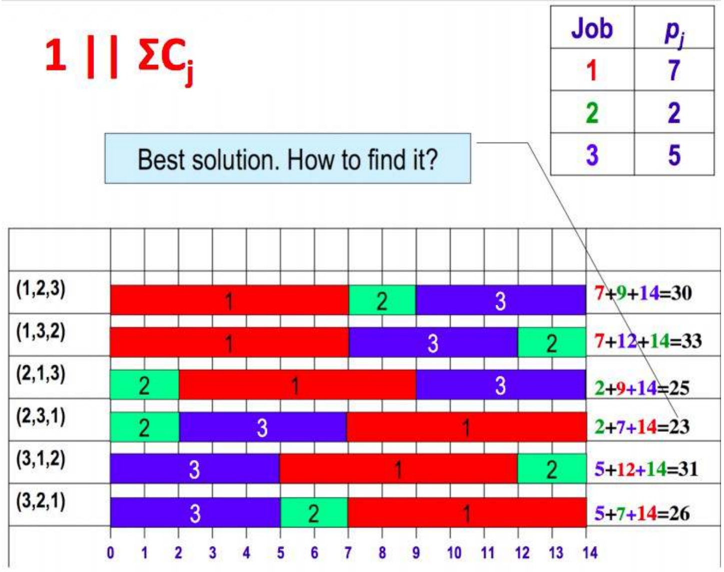
- $C_j$ : completion time of job j
- $L_j = C_j - d_j$ : lateness of job j

**Objectives:**

- Makespan ($\gamma = C_{max}$ ) $C_{max} = \max \{C_1,\ldots,C_n\}$
- Maximum lateness ($\gamma = L_{max}$ ) $L_{max} = \max\{L_1,\ldots,L_n\}$
- Total completion time ($\gamma = \Sigma_j C_j$ )
- Total weighted completion time ($\gamma = \Sigma_j w_jC_j$ )

Many more models in literarture !

*Examples:*




*1| | Sum Cj is solved by ordering jobs in SPT order.*
*(Shortest Production Time)*
*This takes O(n log n) time.*

## 2) $1 \,||\, \Sigma w_j C_j$

| Job | $p_j$ | $w_j$ |
|---|---|---|
| 1 | 1 | 7 |
| 2 | 1 | 2 |
| 3 | 1 | 5 |

Easier case: $1|p_j=1|\ \Sigma w_j C_j$

OPT [1] [3] [2]   $\Sigma w_j C_j = 7 + 5\cdot 2 + 2\cdot 3 = 23$

In an optimal schedule the jobs have to be ordered in **decreasing** (non-increasing) order of their weights.

We have seen:

- If $w_1=w_2=\ldots=w_n$ then smallest jobs go first (SPT).
- If $p_1= p_2=\ldots= p_n$ then largest weight goes first.

For arbitrary $w_j$ and $p_j$ use Smith's ratio rule:

Scheduling in non-increasing order of $w_j/p_j$ (weighted shortest processing time, WSPT) is optimal.

*1 | Sum WjCj is solved by using WSPT order.*
*Weighted Shortest Processing Time)*
*This takes O(n log n) time.*

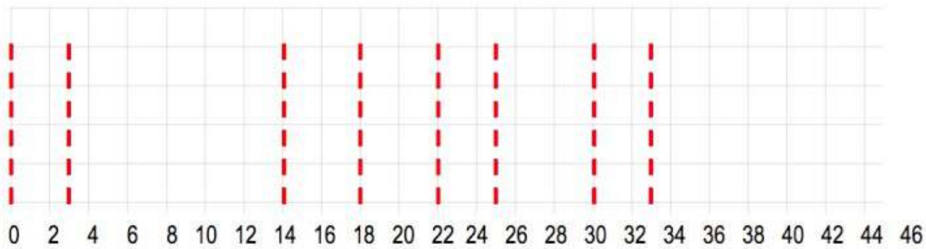
## 3) $1 \mid r_j, pmtn \mid \Sigma C_j$

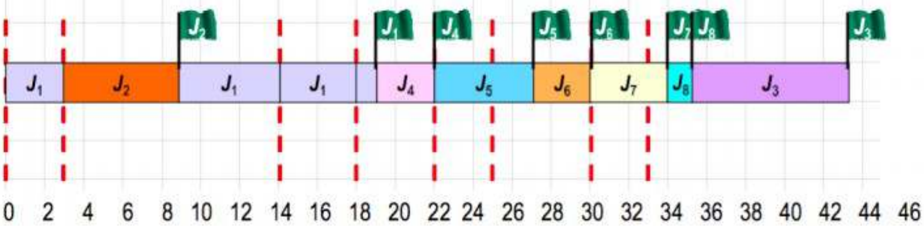
### Shortest Remaning Processing Time rule (SRPT)

At any moment in time, process the job with smallest remaining processing time among the available jobs.

| | $r_j$ | $p_j$ |
|---|---|---|
| $J_1$ | 0 | 13 |
| $J_2$ | 3 | 6 |
| $J_3$ | 14 | 8 |
| $J_4$ | 18 | 3 |
| $J_5$ | 22 | 5 |
| $J_6$ | 25 | 3 |
| $J_7$ | 30 | 4 |
| $J_8$ | 33 | 1 |

***Shortest Remaining Processing Time** first (SRPT) rule:*
each time that a job is completed, or at the next release date, the job to be processed next has the smallest remaining processing time among the available jobs.




## 4) $1 \mid\mid L_{max}$

Minimizing the maximum lateness.

Lateness of job j: $L_j = C_j - d_j$ (=time after the due date)

$L_{max} = \max\{L_1, \ldots, L_n\}$

### Earliest Due Date (EDD)

Schedule jobs in non-decreasing order of due date $d_j$.

## 5) $1 \mid r_j \mid \Sigma C_j$

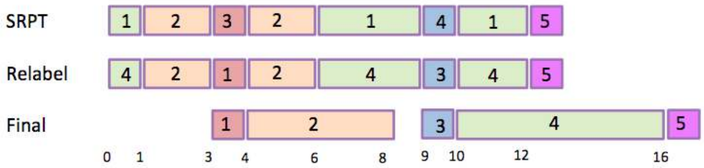
**Algorithm A**

**Step 1**: Apply SRPT. Let $C_1^*, \ldots, C_n^*$ be the completion times.
Assume (relabel) $C_1^* \leq \ldots \leq C_n^*$.

**Step 2**: Schedule jobs in order 1,2,.., n

**Example**

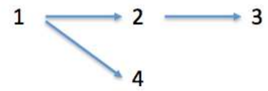
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| rj | 0 | 1 | 3 | 9 | 12 |
| pj | 6 | 4 | 1 | 1 | 1 |

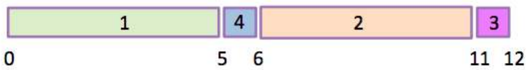



## 6) $1 \mid prec \mid \Sigma C_j$

**Example**

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| pj | 5 | 5 | 1 | 1 |




$\sum Cj = 5+6+11+12 = 34$

**Theorem** The problem $1 \mid prec \mid \Sigma C_j$ is NP-hard (Proof omitted)

*All with 1 machine:*

# Results part 1:

1) $1 \mid\mid \Sigma C_j$ — SPT is optimal

2) $1 \mid\mid \Sigma w_j C_j$ — Smith's ratio rule is optimal: Order by $w_j/p_j$

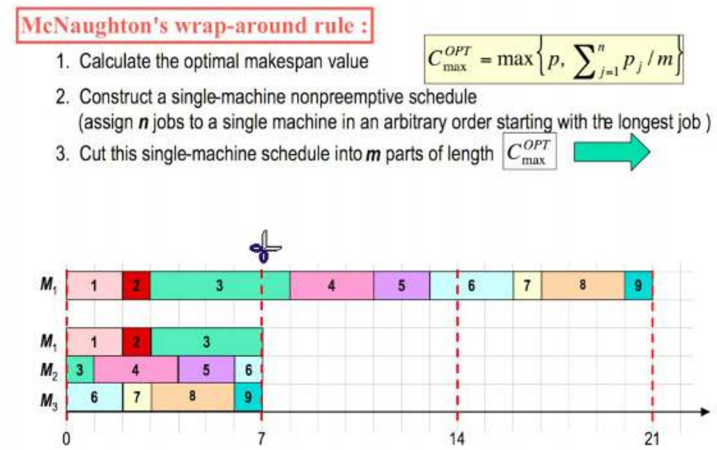
3) $1 \mid r_j, \textit{pmtn} \mid \Sigma C_j$ — SRPT is optimal

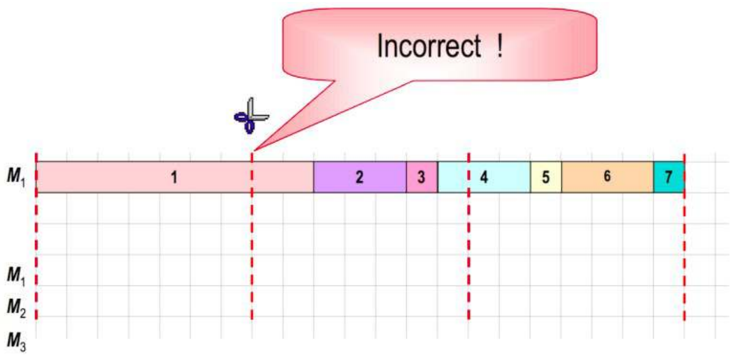
4) $1 \mid\mid L_{max}$ — Earliest Due Date (EDD) is optimal

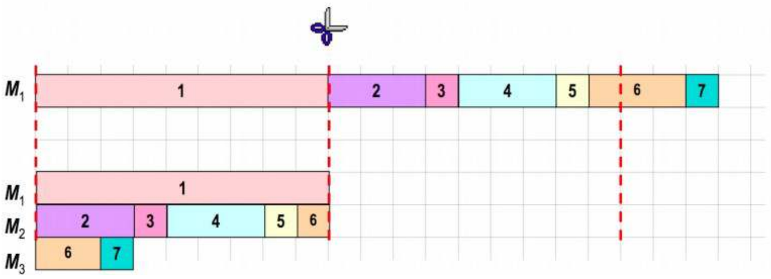
5) $1 \mid r_j \mid \Sigma C_j$ — NP-hard. SRPT order gives 2-approximation.

6) $1 \mid prec \mid \Sigma C_j$ — NP-hard. LP order gives 2-approximation.

## 1) P |pmtn| $C_{max}$



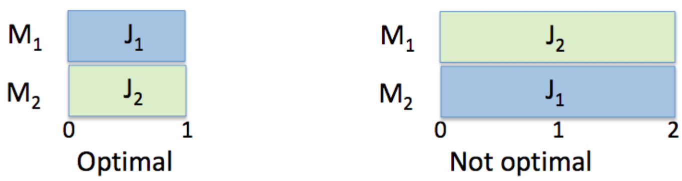




## 3) R | | ∑ $C_j$

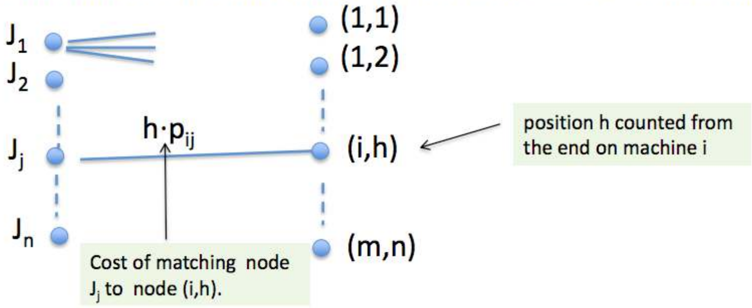
Unrelated machines

$p_{ij}$ : Processing time of job j depends on machine i .

**Example** $p_{11}=p_{22}=1$ and $p_{12}=p_{21}=2$



We can reduce $R \,||\, \sum C_j$ to a minimum cost perfect matching on a complete bipartite graph.
Next, we can reduce this minimum cost perfect matching to a mincost flow problem. (Friday's tutorial)
We know from week 1 that mincost flow can be solved efficiently.




## 4) Rm || $C_{max}$

**Fact** P2 || $C_{max}$ is NP-hard (See exercises this week).
→ Rm || $C_{max}$ is NP-hard too.

The LP-relaxation

$$
\begin{aligned}
\text{(LP)}\ \min\quad & Z \\
s.t.\quad & \sum_{i=1}^{m} x_{ij} = 1 && \text{for all jobs } j \\
& \sum_{j=1}^{n} x_{ij} p_{ij} \le Z && \text{for all machines } i \\
& x_{ij} \ge 0 && \text{for all } i, j
\end{aligned}
$$

**Algorithm:**

Step 1. Solve LP-relaxation
Let $S_1$ be the integer jobs ($x_{ij}=1$)
Let $S_2$ be the fractional jobs (the other jobs).

Step 2. For $S_1$: Assign job j to i if $x_{ij}=1$. Next,
For $S_2$: Try all possible assignments and take the one that gives the smallest makespan ($C_{max}$).

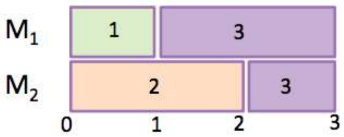
**Algorithm:**

Step 1. Solve LP-relaxation
Let $S_1$ be the integer jobs ($x_{ij}$=1)
Let $S_2$ be the fractional jobs (the other jobs).

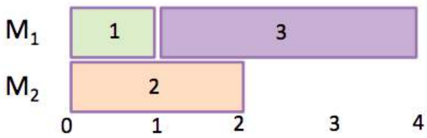
Step 2. For $S_1$: Assign job j to i if $x_{ij}$=1. Next,
For $S_2$: Try all possible assignments and take the one that gives the smallest makespan ($C_{max}$).

**Example:**

| pij | j=1 | j=2 | j=3 |
|---|---|---|---|
| i=1 | 1 | 9 | 5 |
| i=2 | 9 | 2 | 5 |




Optimal LP-solution




Final schedule

**Lemma 1:** Algorithm runs in polynomial time (for m=constant)

- Step 1 LP can be solved in polynomial time.
- Step 2 Claim : There are at most m fractional jobs
  Proof : Next slide

  Given the claim:

  Only $O(m^m)$ possible assignments to check in step 2.
  This is constant for m is constant.

# Results for part 2

**1) P | *pmtn* | $C_{max}$** - McNaughton's wrap arpund rule is optimal.

**2) P | | $C_{max}$**
- NP-hard.
- List scheduling is 2-approximation.
- LPT is 4/3-approximation.

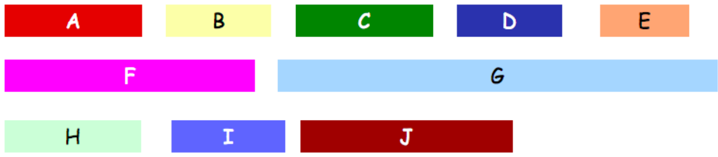
**3) R | | $\Sigma C_j$** - Reducible to min-cost perfect matching.

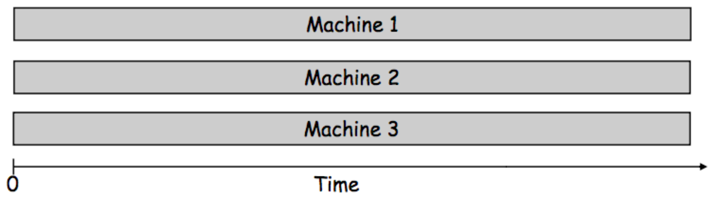
**4) Rm | | $C_{max}$**
- NP-hard.
- LP + enumerating schedules gives 2-approx.

## 2) P || $C_{max}$

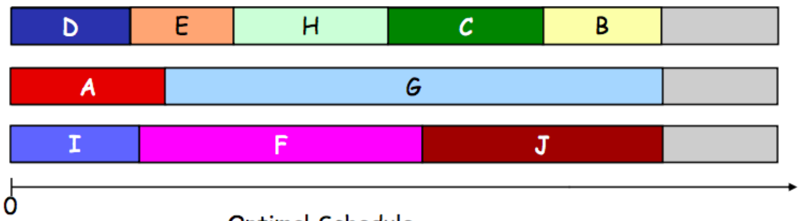
## Minimising $C_{max}$

List Scheduling

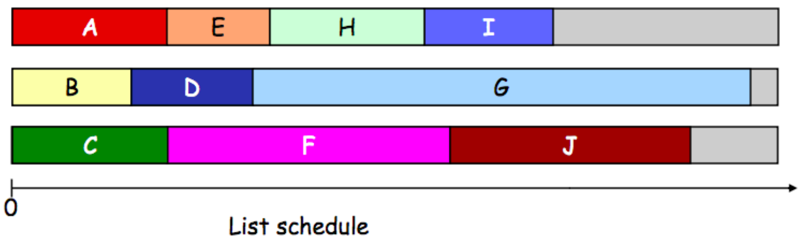



List Scheduling




Optimal Schedule




List schedule

**Longest Processing Time (LPT) rule**

Order jobs by processing time : $p_1 \geq p_2 \geq \ldots \geq p_n$.

Apply list scheduling in this order.

**Theorem:** LPT is 4/3 - approximation for $P||C_{max}$

**Proof :** Friday's tutorial


*Week 5*

*Dynamic Programming*

The idea of DP is always the same: A problems is solved by solving (smaller) subproblems. Solutions to subproblems are stored in memory (the DP table). To solve a subproblem we make use of the stored information on other subproblems. In building and analyzing a DP ask yourself the following questions:

(1) What is the subproblem to solve? In other words: What will be in the table?

(2) What is the optimal value, expressed in terms of the subproblems? In other words: How do you find the optimal value once the table is filled?

(3) What are the initial values? In other words: What values can you fill in right away?

(4) What is the recurrence used? In other words: Given the initial values, how to compute the rest?

(5) What is the used space? This is often the size of the DP table. For example, $O(n)$ or $O(n^2)$. But sometimes we can do with less space, see for example Exercise 1.

(6) What is the running time? This is usually (but not always) the size of the table times the time it takes to compute one value of the table.