# COMP70068 Scheduling and Resource Allocation

Clemen Kok

December 2024

## 1 Introduction to Scheduling

**What is Scheduling?** Job - Unit of work we want to process; Machine - Resource that can process jobs. Sequence - When to start, end, and resume. Also consider parallel workflows, Due dates, Uncertain information.

**Applications of Scheduling.** Comp Arch, Cloud, Production Scheduling, Project Planning.

## 2 Single Machine Scheduling

**Gantt Chart.** How many schedules can we find to sequentially run $n$ jobs without idling times on a single machine - $n!$

**Sequencing.** Optimising functions $f(C_1, ..., C_n)$ - *regular measures* - monotonically non-decreasing with respect to each of the job completion times $C_j, j \leq n$.

Examples of regular measures:

1. $C_{max} = max(C_1, ..., C_2)$ - time at which the last operation is completed (Makespan)
2. $\Sigma C_j$ - Total completion time
3. $\Sigma w_j C_j$ - total weighted completion time
4. $\Sigma w_j(1 - e^{-rC_J})$ - discounted weighted completion time with discount rate $0 \leq r \leq 1$ (e.g. inflation)

**Why would you choose to minimise $C_{max}$?** Minimises Makespan, Total time needed to complete all jobs, Ensures set of jobs is completed ASAP, Ensures system resources are utilised efficiently, maximises throughput, higher cost efficiency, Minimise idleness.

**When to NOT minimise $C_{max}$?** - Cost to idle.

**Classifying Scheduling Problems.** Use notation $a|\beta|\gamma$ with $\alpha$: machines, $\beta$: job characteristics and $\gamma$: optimality criterion.

Assignments:

1. $\alpha = 1$ Single Machine, $P$ Parallel Machines
2. $\beta$: *pmtn* - preemption (interrupts and suspension for higher priorities), $r_j$: release times (time when job becomes available for scheduling), $d_j$: due dates, *prec* precedences
3. $\gamma$: $C_{max}$, $\Sigma C_j$ , $\Sigma w_j C_j$
4. E.g. $1||\Sigma C_j$, $1|r_j, pmtn|\Sigma C_j$, $P|prec|\Sigma w_j C_j$

$PARTITION$**.** Theoretical problem that illustrates the hardness of scheduling for load balancing with known, deterministic, job processing times. Given a set of $S$ integers (e.g. job processing times) with repetitions allowed, $PARTITION$ asks to form two subsets with identical sum, or return that they do not exist. $NP$-hard. With $S = \{1, 1, 1, 2, 2, 2, 3\}$ we may choose $S_1 = \{1, 1, 1, 3\}$ and $S_2 = \{1, 2, 3\}$ or ... or ...

**Proof that $PARTITION$ is $NP$ Complete.** Reduce from Subset Sum Problem.

## 3 Release Times

### Assumptions

- A set of $n$ jobs is immediately ready to be scheduled.
- Deterministic Processing Times, known in advance.
- No preemption.

- Single server machine.
- No explicit setup times.
- No machine idling while jobs are waiting.
  - Idle - non pre-emptive - higher priority.

## Preemptive vs Non-preemptive Scheduling

- No interrupts can lead to improved $C$.
- In single machine scheduling, if performance is monotonic with respect to completion times, there is no advantage in using preemption.

## Scheduling on a Single Resource

- Scheduling on a single resource is the simplest scheduling problem.
- More complex real-world situations can often reduce to analyzing single machine problems when we model the *bottleneck* machine.
- Non-idling schedules: no gains in regular measures by inserting idle times within a single machine. Not true in general outside assumptions.
- $1||C_{\max}$ is trivial: under non-idling, makespan minimization is optimized by any random schedule since:

  $$C_{\max} = \sum_j p_j = \text{const}$$

  where $p_j$ is the processing time of job $j = 1, \ldots, n$.

## Total Completion Time Problem $(1||\sum C_j)$

- Shortest Processing Time (SPT) Rule is optimal.
- Let $[j]$ denote the $j$-th scheduled job, which will have processing time $p_{[j]}$ and completion time $C_{[j]}$. Since all jobs are released at time 0, the latter will be:

  $$\begin{aligned} C_{[1]} &= p_{[1]} \\ C_{[2]} &= p_{[1]} + p_{[2]} \\ &\vdots \\ C_{[n]} &= p_{[1]} + p_{[2]} + \cdots + p_{[n]} \\ \hline \sum C_j &= np_{[1]} + (n-1)p_{[2]} + \cdots + p_{[n]}. \end{aligned}$$

- Shorter jobs are best scheduled earlier.
- Note: job completion time is accumulative.

**Q: Computational Complexity of SPT?** $P|n \log n$

## $1||\sum w_j C_j$ Problem

- Weighted Shortest Processing Time (WSPT) Rule (Smith's rule).
- Schedules jobs in non-decreasing $p_j/w_j$ ratio order.
- Proof follows adjacent pairwise interchange argument:
    - Assume that $S$ is optimal, but not a WSPT schedule. Thus, we can find two adjacent jobs $i$ and $j$ in $S$ such that $\frac{p_i}{w_i} > \frac{p_j}{w_j}$.

## Proof of WSPT Optimality

$$\sum_{l=1}^{n} w_l C_l = w_i\,(p(B) + p_i) + w_j\,(p(B) + p_i + p_j) + \sum_{k \neq i,j} w_k C_k$$

$$\sum_{l=1}^{n} w_l C'_l = w_j\,(p(B) + p_j) + w_i\,(p(B) + p_i + p_j) + \sum_{k \neq i,j} w_k C_k$$

$$\sum w_l C_l - \sum w_l C'_l = w_j p_i - w_i p_j > 0$$

Since $S - S' > 0$, $S > S'$.

**Key Idea:** Swapping two jobs in the sequence affects completion time proportional to weights and processing times.

## Job Chains as an Extension of WSPT

- Sequential precedences among jobs.
- Non-interruptible execution.
- Let:

$$A : 1 \to 2 \to \cdots \to k$$

$$B : k+1 \to k+2 \to \cdots \to n.$$

  Which should be processed first?

- $A$ should be scheduled before $B$ if:

$$\frac{\sum_{j=1}^{k} p_j}{\sum_{j=1}^{k} w_j} < \frac{\sum_{j=k+1}^{n} p_j}{\sum_{j=k+1}^{n} w_j}$$

- Similar to WSPT (non-decreasing chain ratios).

**Interruptible Execution:**

- Interleave executions of jobs from different chains.
- Individual jobs are still executed non-preemptively.

- WSPT may still be applied to sub-chains.

**$\rho$-factor:**

$$\rho_A = \min_{1 \le l \le k} \frac{\sum_{j=1}^{l} p_j}{\sum_{j=1}^{l} w_j}.$$

- Initial subchains of $A$ of length $l$: the one with the minimum ratio determines the $\rho$ factor.

### Uncertain Processing Times

- Consider $1||\sum C_j$ and $1||\sum w_j C_j$ with uncertain $p_i$.
- Relaxes assumption A2.
- Assume each processing time $p_i$ is sampled from a statistical distribution, making $p_i$ a random variable.
- Model assumption A5 (Machine Breakdowns): A breakdown could unexpectedly extend the processing time of the job in service until it resumes after machine repair.

**Key Points:**

- Maximum is not tractable in stochastic vs deterministic cases.
- Optimize expected total completion time $E[\sum C_j]$.
- SPT can be replaced by S-Expected-PT policy, scheduling jobs in order of mean processing time.

Let $p_j$ be independent random variables:

$$E[\sum C_j] = E\left[np_{[1]} + (n-1)p_{[2]} + \cdots + p_{[n]}\right] = nE[p_{[1]}] + (n-1)E[p_{[2]}] + \cdots + E[p_{[n]}].$$

If $p_j$ are exponentially distributed with rates $\mu_j = 1/E[p_j]$ and weights are $c_j$, the principle is known as the $c\mu$ rule.

## 4 Due Dates

### Key Definitions

Each job has a due date $d_j$ (delivery time). Due dates are soft and can be violated. The goal is to minimise deviations. The following metrics are defined:

- **Lateness:** $L_j = C_j - d_j$
- **Tardiness:** $T_j = \max(0, C_j - d_j)$

## Minimising Total Lateness $(1||\Sigma L_j)$

Although the Shortest Processing Time (SPT) rule ignores the due dates $d_j$, it remains optimal for minimising total lateness.

$$\Sigma_j L_j = \Sigma_j (C_j - d_j) = \Sigma_j C_j - \Sigma_j d_j$$

Since $D = \Sigma_j d_j$ is given and independent of the schedule, minimising total lateness is equivalent to minimising total completion time. A similar situation arises for the weighted analogue $1||\Sigma w_j L_j$.

## Focusing on $L_{\max}$

The total lateness $\Sigma L_j$ does not fully capture the significance of due dates. Hence, the problem $1||\Sigma L_j$ is seldom studied. Instead, the focus is often on minimising the maximum lateness $L_{\max}$:

$$L_{\max} = \max_j L_j$$

## Earliest Due Date (EDD) Rule

For jobs with different due dates $d_i \geq d_j$, the **Earliest Due Date (EDD)** rule, which schedules jobs in non-decreasing $d_j$ order, is optimal for $1||L_{\max}$. Ties are broken at random.

EDD is also optimal for $1||T_{\max}$. The relationship between $T_{\max}$ and $L_{\max}$ is:

- $T_{\max} = 0$ when $L_{\max} \leq 0$
- $T_{\max} = L_{\max}$ if $L_{\max} > 0$

## Proof of EDD Optimality

Proof details can be found in the notes.

## Stochastic Case

Under uncertainty, lateness and tardiness are harder to handle due to expectations involving maxima of random variables. However, it can still be shown that EDD remains optimal for $1||L_{\max}$ and $1||T_{\max}$.

## Minimising Total Tardiness $(1||\Sigma T_j)$

Minimising $\Sigma T_j$ provides more control over tardiness distribution across jobs. However:

- The problem is NP-hard.

- No optimal rule like EDD or SPT exists.
- EDD is still optimal in restricted cases.

### Agreeable Jobs

Jobs are **agreeable** if, for any two jobs $i$ and $j$, $p_i \geq p_j$ implies $d_i \geq d_j$. If all jobs are agreeable, total tardiness is minimised by EDD sequencing with ties broken by SPT. This can be proven using adjacent pairwise interchange analysis.

## Minimising Number of Tardy Jobs ($\Sigma_j U_j$)

The number of tardy jobs is defined as:

$$U_j = \begin{cases} 0 & \text{if } C_j \leq d_j \\ 1 & \text{otherwise} \end{cases}$$

### Moore-Hodgson Algorithm

The problem $1||\Sigma_j U_j$ is solvable exactly in polynomial time using the **Moore-Hodgson Algorithm**. However, the weighted case is NP-hard.

1. Jobs $j = 1, \ldots, n$ are added one by one, in EDD order, to the on-time schedule $S_O$.
2. If job $j$ completes after time $d_j$, the job $i \in S_O$ with the largest $p_i$ is declared late and moved to the late schedule $S_L$ in an arbitrary position.
3. Step 2 is repeated until either job $j$ can be added to $S_O$ or it is moved to $S_L$ if it has the largest $p_i$.
4. After all jobs are added, the algorithm returns $S = \{S_O, S_L\}$:
   - On-time jobs ($S_O$) are scheduled before late jobs ($S_L$).
   - The order of jobs in $S_L$ is arbitrary.

**Example:**

- Remove $J_2$, then $J_1$ (by processing time $p_i$).
- These jobs will be late regardless.

# 5 Enumeration and Local Search

## Introduction

Many scheduling problems are NP-Hard and cannot be solved efficiently if the problem size is large enough. Combinatorial optimization involves minimizing a cost function $g(S)$ over all feasible schedules $S$.

In most cases, the cost function is an additive function:

$$g(S) = \sum_j g_j(S),$$

where $g_j(S)$ is the cost of scheduling job $j$ prescribed by schedule $S$. - Example: $g_j(S) = w_j U_j$ defines a $1||\sum_j w_j U_j$ problem. - Example: $g_j(S) = \max(0, C_j - d_j)$ defines a $1||\sum_j T_j$ problem.

## Local Search

Similar to hill climbing but applied to discrete problems. A neighborhood $\mathcal{N}(S)$ of the current solution $S$ is generated using adjacent pairwise interchange, subject to constraints. The solution evolves by selecting the next one in $\mathcal{N}(S)$. May get trapped in local optima.

## Simulated Annealing (SA)

A random-walk-based search that explores the solution space while being guided towards the global optimum over time. Escape tendency around local optima is controlled by a temperature parameter $T_k$, reduced over time until convergence.

Algorithm:

1. Start with an initial solution $x_0$.
2. At iteration $k+1$, randomly choose a neighbour $y \in \mathcal{N}(x_k)$.
3. Compute cost change $\Delta = g(x_k) - g(y)$: If $\Delta \geq 0$, accept $y$ as $x_{k+1} = y$. If $\Delta < 0$, accept $y$ with probability $e^{\Delta/T_k}$.
4. Update temperature:

$$T_{k+1} = \alpha T_k = \alpha^{k+1} T_0, \quad T_0 > 0, \quad 0 < \alpha < 1.$$

5. Stop when $k > K$.

Alternative cooling model:

$$T_k = \frac{T_0}{1 + \log(1+k)}.$$

## Tabu Search

A deterministic global optimization method using a memory structure called the tabu list $\mathcal{T}$. Tracks recently swapped job pairs to avoid cycling back to previous solutions. Exceptions (aspiration criteria) allow revisiting a solution if it improves the best overall result.

Algorithm:

1. Swap adjacent jobs $i$ and $j$ in schedule $S$. Add pair $(i, j)$ to $\mathcal{T}$.
2. Prevent re-swapping $(i, j)$ for $L$ iterations unless aspiration criteria are met.
3. Stop when no further improvements are possible.

## Handling Constraints

Schedules may have equality and inequality constraints:

$$g_{\text{best}} = \min_{S} g(S), \quad \text{s.t. } f_i(S) = 0,\ h_k(S) \leq 0.$$

### Rejection Method

Skip candidate solutions violating constraints. Valid initial schedules required.

### Penalty Function Method

Modify the cost function:

$$g'(S) = g(S) + \sum_{i=1}^{I} \lambda_i f_i(S) + \sum_{k=1}^{K} \gamma_k \max(0, h_k(S)).$$

## Dynamic Programming (DP)

Define $J$ as a set of jobs. Let $G(J)$ be the minimum cost of scheduling the jobs in $J$, where $G(J)$ is obtained recursively:

$$G(J) = \min_{j \in J} \{G(J - \{j\}) + g_j(J)\}, \quad G(0) = 0.$$

Complexity: $O(n2^n)$.

## Branch and Bound

Combines branching (partitioning solutions) and bounding (eliminating suboptimal solutions). Nodes in a search tree represent subproblems with specific jobs fixed in the schedule. Subproblems are pruned using bounds to save computation.

# 6 Parallel Machine Scheduling

## Introduction

1. Jobs are mapped to $m$ identical parallel machines. 2. A job cannot run on two machines at the same time. 3. The makespan problem, denoted $P||C_{max}$, is no longer trivial and equates to balancing job loads across machines. 4. For $m = 2$, the problem is denoted $P_2$ (parallel scheduling on two machines).

## Total Completion Time Problems

1. The Shortest Processing Time (SPT) optimality for $1||\sum C_j$ can be generalized to $P||\sum C_j$. 2. Adding indices to represent completion times and processing on $m$ machines, the goal is to minimize:

$$\begin{aligned}\sum_{j=1}^{n_1} C_{1,j} &= n_1 p_{1,[1]} + (n_1 - 1)p_{1,[2]} + \cdots + p_{1,[n_1]},\\ \vdots \quad & \quad \vdots \\ \sum_{j=1}^{n_m} C_{m,j} &= n_m p_{m,[1]} + (n_m - 1)p_{m,[2]} + \cdots + p_{m,[n_m]}.\end{aligned}$$

3. SPT optimizes the system by sorting jobs in ascending size and then applying round-robin across machines. 4. Weighted Shortest Processing Time (WSPT) assigns jobs to the first idle machine in increasing $p_i/w_i$ ratios. While not optimal for $P||\sum w_j C_j$, which is NP-hard, it serves as an approximation with a worst-case ratio:

$$R = \frac{1+\sqrt{2}}{2} \approx 1.21.$$

5. Round-robin schedules jobs cyclically through $m_1, m_2, \ldots, m_n$.

## Makespan Problems

1. With $m$ machines, the makespan admits the lower bound:

$$M^\star = \max(p_{max}, \frac{\sum p_j}{m}),$$

where $p_{max} = \max p_j$. 2. The term $\sum p_j/m$ represents the optimal case where all machines finish jobs simultaneously. 3. The $p_{max}$ term arises because no job can run on multiple machines simultaneously, so the longest job determines a minimum makespan.

## McNaughton’s Wrap Around Rule

1. Schedule an arbitrary job on machine $m_1$ at time 0. 2. Start any unscheduled job as soon as possible on the same machine. 3. Repeat the process until the makespan on a machine exceeds $M^\star$ or all jobs are scheduled. 4. Reassign processing beyond $M^\star$ to the next machine, starting at time 0. Repeat the process.

## Non-Preemptive Case

1. The non-preemptive makespan problem is NP-hard, even with $m = 2$ machines. 2. Assuming integer $p_j$ and $\sum p_j$ divisible by 2, $P_2||C_{max}$ asks to partition integers into two subsets summing to $M^\star$, equivalent to the partition problem.

## List Scheduling (LS)

1. List scheduling is a popular approximation. Given a priority list of jobs, LS assigns the first available job to the first available machine. 2. The worst-case ratio for $P||C_{max}$ is:

$$R = 2 - \frac{1}{m}.$$

### Proof of the Worst-Case Ratio

1. Let $t_k = C_{max}^{LS} - p_k$ be the time when job $k$ starts. Since machines are always busy before $t_k$:

$$m \cdot t_k \leq \sum_{j=1}^{n} p_j - p_k.$$

2. Substituting $t_k = C_{max}^{LS} - p_k$ and rearranging:

$$C_{max}^{LS} \leq \frac{\sum_{j=1}^{n} p_j}{m} + \frac{(m-1)p_k}{m}.$$

3. The optimal makespan satisfies $C_{max}^{OPT} \leq M^\star$, where:

$$\frac{\sum_{j=1}^{n} p_j}{m} \leq C_{max}^{OPT}, \quad p_k \leq p_{max} < C_{max}^{OPT}.$$

4. Therefore:

$$\frac{C_{max}^{LS}}{C_{max}^{OPT}} = 2 - \frac{1}{m}.$$

### Longest Processing Time (LPT)

1. List scheduling achieves a $(2 - \frac{1}{m})$ approximation if no specific job priority order is specified. 2. LPT assigns jobs in non-increasing order of size. LS with LPT achieves a smaller worst-case ratio:

$$R = \frac{4}{3} - \frac{1}{3m}.$$

## 7 Workflow Scheduling

### Directed Acyclic Graphs (DAGs)

1. Job precedences are represented as a Directed Acyclic Graph (DAG) $G = (V, E)$:
    - Vertices $V$ are jobs.
    - An edge $(i, j) \in E$ indicates that job $i$ must complete before job $j$ can start.
2. Precedence constraints frequently require forced idleness.
3. Scheduling theory under precedence constraints is challenging:
    - Most problems with precedences are NP-hard.
    - Complexity remains an open question in some cases.
4. Optimal methods exist only under restrictive assumptions:
    - Unit processing times ($p_j = 1, \forall j$).
    - Fixed number of processors (e.g., two).
    - Restricted topologies (chains, in-trees, out-trees):
        - In-tree: Every node has at most one child.
        - Out-tree: Every node has at most one parent.
        - Chain: Every node has at most one parent and one child.

### Hu's Algorithm

1. Hu's algorithm solves $P|i.t., p_j = 1|C_{max}$ (parallel scheduling with in-tree precedence constraints).
2. Assign a level $\alpha_j$ to each job $j$ as follows:
    - The exit node is labeled with $\alpha_j = 1$.
    - For all other nodes:
    $$\alpha_j = 1 + \max_{i:(j,i)\in E} \alpha_i.$$

3. The maximum level is $L = \max_i \alpha_i$, and $\alpha_j - 1$ is the path length from $j$ to the exit node. The longest path is called the critical path.
4. Hu's algorithm schedules ready jobs in non-decreasing $\alpha_j$ order, with complexity $\mathcal{O}(n)$.
5. The algorithm achieves optimal schedules for $P|i.t., p_j = 1|C_{max}$.

## Optimality of Hu's Algorithm

1. Reduction to in-tree cases ensures optimality:
   - Reverse the arc orientations in out-tree problems to create an equivalent in-tree problem.
   - For disjoint in-trees, add a dummy task as the successor of all exit nodes.
2. The algorithm approximates $P|prec, p_j = 1|C_{max}$:
   - Worst-case ratio:

$$R = \begin{cases} \frac{4}{3}, & \text{if } m = 2, \\ 2 - \frac{1}{m-1}, & \text{if } m \geq 3. \end{cases}$$

## Muntz-Coffman Algorithm

1. The Muntz-Coffman algorithm extends Hu's algorithm to the preemptive case.
2. A subset sequence $S_1, S_2, \ldots, S_k$ is defined as follows:
   - Each job $a \in G$ belongs to some $S_i$.
   - If $a \in S_j$ is a successor of $b \in S_i$, then $j > i$.
3. Given a subset sequence, schedule subsets in increasing $i$:
   - If $|S_j| > m$, use McNaughton's wrap-around rule, assigning $|S_j|/m$ time on $m$ processors.
   - If $|S_j| \leq m$, assign one unit of time on $|S_j|$ processors.
4. The method is optimal for general DAGs, subject to $m = 2$ and $p_j = 1$.

## Optimality of Muntz-Coffman Algorithm

1. The algorithm is optimal for:
   - $P_2|pmtn, prec, p_j = 1|C_{max}$.
   - $P|pmtn, i.t., p_j = 1|C_{max}$.

- $P|pmtn, o.t., p_j = 1|C_{max}$.

2. Under preemptive scheduling, $p_j > 1$ can be divided into jobs with $p_j = 1$.
3. The worst-case ratio for general preemptive scheduling $P|pmtn, prec|C_{max}$ is:
$$R = 2 - \frac{2}{m}, \quad m \geq 2.$$
4. The algorithm is optimal for $m = 2$ and near-optimal for small $m > 2$.

## 8 Bottleneck Analysis

### System Overview

1. Jobs arrive at the system, are processed by machines, and then completed.
2. Assumptions:
   - A1: The system has $M$ machines with arbitrary speeds.
   - A2: Jobs arrive at arbitrary times, not known in advance.
   - A3: Jobs may visit more than one machine and do not incur communication overheads.
   - A4: Job processing times are arbitrary and machine-dependent.
   - A5: Job sequencing at machines is arbitrary.
   - A6: Inside the system, there is no job creation, destruction, or parallel execution of a single job.
3. Key questions:
   - Bottleneck Analysis: Which machines limit the peak completion rate?
   - What-if Analysis: How will changes in job arrival rates or machine speeds impact the system?
4. Performance factors:
   - Arrival rate of jobs.
   - Processing time of jobs at the machine.
   - Utilization (fraction of time the machine is busy).
   - Contention.

## Job Classes

1. The open system:
   - Processes jobs using $M$ machines.
   - Offers $C$ types of services (job classes).
   - Receives class-$c$ jobs at rate $\lambda_c$.
2. For a class-$c$ job:
   - Visits machine $i$, on average, $v_{ic}$ times.
   - Requires a mean processing time $p_{ic}$ for each visit.
   - Demand:
     $$D_{ic} = v_{ic} p_{ic},$$
     the total processing time accumulated on average during visits to machine $i$.

## Operational Analysis

1. Monitor a machine for an observation period $T$ seconds and collect:
   - $A_c$: Total number of arrived jobs of class $c$.
   - $B_{ic}$: Total time machine $i$ is busy processing class-$c$ jobs.
   - $F_c$: Total number of finished jobs of class $c$.
2. Compute:
   - $\lambda_c = A_c/T$: Average arrival rate of jobs of class $c$.
   - $X_c = F_c/T$: Average system throughput of jobs of class $c$.
   - $U_{ic} = B_{ic}/T$: Utilization of machine $i$ for class-$c$ jobs.
3. Stability:
   - If the number of pending jobs remains finite, the system is stable.
   - In a stable system:
     $$\lambda_c = X_c = \lim_{T \to \infty} \frac{F_c}{T}.$$
   - Unstable systems cannot cope with arrival rates, causing the backlog of pending jobs to grow unbounded.

## Utilization Law

1. Demand:
$$D_{ic} = \frac{B_{ic}}{A_c}.$$
2. Utilization:
$$U_{ic} = \frac{B_{ic}}{T} = \frac{A_c}{T} \cdot \frac{B_{ic}}{A_c} = \lambda_c D_{ic}.$$
3. For $C$ job classes, the total utilization of machine $i$ is:
$$U_i = \sum_{c=1}^{C} U_{ic} = \sum_{c=1}^{C} \lambda_c D_{ic} = \sum_{c=1}^{C} X_c D_{ic}.$$
4. Estimating demand:
   - Use multivariate linear regression to fit hyperplanes to samples of $U_i$ and $\lambda_c$.
   - Estimated demands $D_{ic}$ are hardware-dependent and change after machine upgrades.

## Bottleneck Analysis

1. Bottlenecks:
   - Machines that limit performance by struggling to handle arrival rates.
   - Tend to operate near 100
2. Single class ($C = 1$):
   - For every machine $i$:
   $$U_i = \lambda D_i.$$
   - Maximum arrival rate:
   $$\lambda \leq \frac{1}{D_{\max}},$$
   where $D_{\max} = \max(D_1, \ldots, D_M)$.
   - Bottlenecks correspond to machines with $D_{\max}$.
3. Multi-class ($C > 1$):
   - Machine usage is described by:
   $$U_i = \sum_{c=1}^{C} \lambda_c D_{ic}.$$
   - A machine $j$ can saturate if there exists a mix of arrival rates $\lambda_1, \ldots, \lambda_C$ such that $U_j = 1$.

- Verify saturation using linear programming (LP):

$$U_j^{\max} = \max \sum_{c=1}^{C} \lambda_c D_{jc},$$

subject to:

$$\sum_{c=1}^{C} \lambda_c D_{ic} \leq 1, \quad \lambda_c \geq 0.$$

- If $U_j^{\max} = 1$, the machine can saturate.

# 9 Competitive Decision Making

## Introduction

1. Multi-agent settings lack a clear notion of what is optimal.
   - Focus is on equilibria rather than global optimizers.
   - An equilibrium is "good" if it overlaps with system optima.
2. Game setup:
   - Players: $i = 1, \dots, N$.
   - Each player has a set of actions $x_i \in \mathcal{X}_i$.
   - Each player incurs a cost $J_i(x_1, \dots, x_N)$, which depends on everyone's choices.
3. Example: The 2/3 game.

$$J_i(x_1, \dots, x_N) = \left| \frac{2}{3} \cdot \frac{1}{N} \sum_j x_j - x_i \right|.$$

4. Bimatrix representation example:

| | $R$ | $S$ |
|---|---|---|
| $R$ | $(30, 0)$ | $(30, 10)$ |
| $S$ | $(100, 100)$ | $(0, 10)$ |

## Nash Equilibria

1. A Nash equilibrium (N.E.) models emergent behavior.
   - A pure N.E. is a feasible allocation such that no player can decrease their cost by unilateral deviation.

2. Definition:

   $(x_1^\star, \dots, x_N^\star) \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_N$ is a N.E. if $J_i(x_i^\star, x_{-i}^\star) \le J_i(x_i, x_{-i}^\star) \quad \forall x_i \in \mathcal{X}_i, \forall i.$

   Here:

   - $x_{-i}$ represents the strategies of all players except $i$.
   - $x_i$ is the strategy of player $i$.

3. Example:

   | | $A$ | $B$ |
   |---|---|---|
   | $A$ | $(30, 0)$ | $(30, 10)$ |
   | $B$ | $(11, 11)$ | $(0, 10)$ |

   The N.E. is $(10, 10)$, as neither player has an incentive to deviate.

## Best Response Algorithm

1. Given other players' strategies $x_{-i}$, the best response is:

$$BR(x_{-i}) = \arg \min_{x_i \in \mathcal{X}_i} J_i(x_i, x_{-i}).$$

2. Steps:

   (a) Initialize strategies $x$ and set $i = 1$.

   (b) Check if $x$ is an equilibrium; if so, stop.

   (c) Update $x_i \leftarrow BR(x_{-i})$.

   (d) Set $i \leftarrow (i \mod N) + 1$.

   (e) Repeat.

3. Convergence:

   - If the current allocation minimizes the cost, no one will change their strategy.
   - The algorithm converges to a Nash equilibrium.

## Mixed Nash Equilibria

1. Players may randomize their actions, minimizing expected cost.

2. For finite actions, let $\sigma_i \in \Delta_i$ represent the probabilities of player $i$'s actions. The expected cost is:

$$C_i(\sigma_1, \dots, \sigma_N) = \mathbb{E}_{x \sim \sigma}[J_i(x)] = \sum_{x \in \mathcal{X}} p(x) J_i(x).$$

3. Definition:

   $(\sigma_1^\star, \dots, \sigma_N^\star) \in \Delta_1 \times \cdots \times \Delta_N$ is a mixed N.E. if $C_i(\sigma_i^\star, \sigma_{-i}^\star) \le C_i(x_i', \sigma_{-i}^\star) \quad \forall x_i' \in \mathcal{X}_i, \forall i.$

### Matching Pennies Example

1. Game:

|  | $H$ | $T$ |
|---|---|---|
| $H$ | $(1,-1)$ | $(-1,1)$ |
| $T$ | $(-1,1)$ | $(1,-1)$ |

2. No pure Nash equilibria exist.
3. Mixed Nash equilibria:
   - Let $p$ be the probability the row player chooses $H$.
   - Let $q$ be the probability the column player chooses $H$.
4. Expected payoffs:

$$\text{Row Player (P1):} \begin{cases} H : 1 \cdot q + (-1) \cdot (1-q) = 2q - 1, \\ T : (-1) \cdot q + 1 \cdot (1-q) = -2q + 1. \end{cases}$$

$$\text{Column Player (P2):} \begin{cases} H : -1 \cdot p + 1 \cdot (1-p) = -2p + 1, \\ T : 1 \cdot p + (-1) \cdot (1-p) = 2p - 1. \end{cases}$$

5. Solve for indifference:

$$2q - 1 = -2q + 1 \implies q = \frac{1}{2},$$

$$-2p + 1 = 2p - 1 \implies p = \frac{1}{2}.$$

6. The mixed Nash equilibrium is $(p, q) = \left(\frac{1}{2}, \frac{1}{2}\right)$.
7. Expected cost:

$$\mathbb{E}[C_1] = \frac{1}{2} \cdot \frac{1}{2} - \frac{1}{2} \cdot \frac{1}{2} - \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = 0.$$

   Similarly, $\mathbb{E}[C_2] = 0$.

8. Deviation:
   - If one player deviates, the expected cost remains 0.

## 10 Potential and Congestion Games

### Potential Games

1. A potential game is a "nice" game where the cost functions of all players can be described by a single potential function.

2. Formal definition:

   - A strategic game is a potential game if there exists a function $\Phi : \mathcal{X} \to \mathbb{R}$ such that:

     $$J_i(x_i, x_{-i}) - J_i(y_i, x_{-i}) = \Phi(x_i, x_{-i}) - \Phi(y_i, x_{-i}),$$

     for all $x_i, y_i, x_{-i}$, and for all players $i$.

3. The differences in the potential function, rather than its absolute values, determine the dynamics.

4. A potential game guarantees:

   - Existence of at least one Nash Equilibrium (NE).
   - Convergence of the Best Response (BR) algorithm in a finite number of steps for games with finite actions.

5. Intuition:

   - Each time a player improves their utility, the potential changes accordingly.
   - Cyclic behavior is impossible as it would imply $\Phi(x^1) > \Phi(x^2) > \cdots > \Phi(x^1)$, which is contradictory.

## Congestion Games

1. Congestion games model scenarios where players share resources, and the cost depends on resource usage.

2. Components:

   - Set of resources $\mathcal{R}$.
   - Resource cost functions $\updownarrow_r(\cdot)$: latency functions specific to each resource.
   - Set of players $\{1, \ldots, N\}$.
   - Feasible set $\mathcal{X}_i \subseteq 2^{\mathcal{R}}$ for each player $i$, representing the resources they can choose.
   - Player cost:

     $$J_i(x) = \sum_{r \in x_i} \updownarrow_r(|x|_r),$$

     where $|x|_r$ is the number of players using resource $r$.

3. Example: Load balancing.

   - Resources are machines.
   - Players are tasks managed by players.

- Each player's cost equals the runtime of their chosen machine.

4. Best Response (BR) convergence:
    - Congestion games are potential games, with potential function:

$$\Phi(x) = \sum_{r \in \mathcal{R}} \sum_{j=1}^{|x|_r} \updownarrow_r(j).$$

    - The BR algorithm converges because the potential decreases with each step.

## Convergence in Singleton Congestion Games

1. Definition:
    - A singleton congestion game is a congestion game where each player chooses a single resource ($|x_i| = 1$ for all $x_i \in \mathcal{X}_i$).
2. Example:
    - Load balancing games are singleton congestion games because each task can choose only one machine.
3. Convergence:
    - In singleton congestion games with $n$ players and $m$ resources, BR converges in $O(n^2 m)$.
4. Proof intuition:
    - Replace original latencies with bounded integer values while preserving player preferences.
    - The potential $\Phi(x)$ measures total latency across all resources:

$$\Phi(x) = \sum_{r=1}^{m} \sum_{j=1}^{|x|_r} \bar{\updownarrow}_r(j),$$

      where $\bar{\updownarrow}_r(j)$ are the sorted integer latencies for resource $r$.
    - The potential is bounded by $n^2 m$, as the maximum decrease in potential per BR step is at least 1.
5. Example calculation:
    - Suppose $R_1$ and $R_2$ are resources with latencies:

$$\updownarrow_{R_1}(1) = 1, \quad \updownarrow_{R_1}(2) = 3, \quad \updownarrow_{R_1}(3) = 5,$$
$$\updownarrow_{R_2}(1) = 2, \quad \updownarrow_{R_2}(2) = 4, \quad \updownarrow_{R_2}(3) = 6.$$

      After sorting latencies, assign new integer values:

$$\bar{\updownarrow}_{R_1}(1) = 1, \quad \bar{\updownarrow}_{R_2}(1) = 2, \quad \bar{\updownarrow}_{R_1}(2) = 3.$$

- Compute potential:
  - $R_1$ has 2 players. Sum latencies:
    $$\bar{\updownarrow}_{R_1}(1) + \bar{\updownarrow}_{R_1}(2) = 1 + 3 = 4.$$
  - $R_2$ has 1 player. Sum latency:
    $$\bar{\updownarrow}_{R_2}(1) = 2.$$
  - Total potential:
    $$\Phi(x) = 4 + 2 = 6.$$
- The potential decreases with each BR step and is bounded by $n^2m$.

## 11 Efficiency of Equilibria

### Braess Paradox

1. Consider two routes from $A$ to $B$:
   - North: road and ferry.
   - South: ferry and road.
2. Initially, 200 players split equally between the two routes $(100, 100)$.
3. Introducing a new bridge changes the equilibrium:
   - Everyone avoids the ferry, resulting in:
     $$J_i(x) = 2(15 + 0.1 \cdot 200) = 70, \quad \forall i.$$
   - Using the road and ferry leads to a higher cost:
     $$15 + 0.1 \cdot 200 + 400 = 75.$$
4. The new equilibrium increases overall travel cost, illustrating Braess Paradox.

### Price of Anarchy (PoA)

1. Definition:
   - In a strategic game, a social cost function $SC : \mathcal{X} \to \mathbb{R}$ measures the quality of each allocation for the whole population.
   - The price of anarchy is defined as:
     $$PoA = \frac{\max_{x \in PNE} SC(x)}{\min_{x \in \mathcal{X}} SC(x)}.$$

2. Interpretation:
    - Measures the performance degradation from selfish decision-making.
    - $SC(x)$ is typically the sum of players' costs.
3. Example:
    - Worst Nash equilibrium (no ferry): $SC = 14000$.
    - Optimal allocation: $SC = 12875$.
    - Price of anarchy:
$$PoA = \frac{14000}{12875} \approx 1.087.$$
4. Implication:
    - PoA quantifies the inefficiency caused by selfish behavior.
    - Use incentives to reduce $PoA$.

## Bounding the Price of Anarchy

1. Definition of a smooth game:
    - A strategic game is $(\lambda, \mu)$-smooth if there exist constants $\lambda > 0$ and $\mu < 1$ such that:
$$\sum_i J_i(x_i', x_{-i}) \leq \lambda SC(x') + \mu SC(x), \quad \forall x', x \in \mathcal{X}.$$
2. PoA bound:
    - In a $(\lambda, \mu)$-smooth game:
$$PoA \leq \frac{\lambda}{1-\mu}.$$
3. Proof:
    - Let $x$ be a Nash equilibrium and $x'$ an optimal allocation.
    - From the definition of Nash equilibrium:
$$SC(x) = \sum_i J_i(x) \leq \sum_i J_i(x_i', x_{-i}).$$
    - Using the smoothness property:
$$\sum_i J_i(x_i', x_{-i}) \leq \lambda SC(x') + \mu SC(x).$$
    - Combining:
$$SC(x) \leq \lambda SC(x') + \mu SC(x).$$
    - Rearranging:
$$SC(x)(1-\mu) \leq \lambda SC(x'),$$
$$\frac{SC(x)}{SC(x')} \leq \frac{\lambda}{1-\mu}.$$

### Affine Congestion Games and Exact PoA

1. For congestion games with affine latencies:
$$\ell_r(|x|_r) = \alpha_r |x|_r + \beta_r.$$
2. Every such game is $\left(\frac{5}{3}, \frac{1}{3}\right)$-smooth, giving:
$$PoA \leq \frac{5}{2} = 2.5.$$
3. Proof (outline):
   - Social cost:
$$SC(x) = \sum_i J_i(x) = \sum_r |x|_r \ell_r(|x|_r).$$
   - Smoothness condition:
$$\sum_i J_i(x'_i, x_{-i}) \leq \frac{5}{3} SC(x') + \frac{1}{3} SC(x).$$
   - Goal:
$$|x'_r| \ell_r(|x|_r + 1) \leq \frac{5}{3} |x'_r| \ell_r(|x'_r|) + \frac{1}{3} |x_r| \ell_r(|x|_r), \quad \forall x, x', r.$$

### Congestion Pricing

- Design tolls $\tau_r(|x|_r)$ for each resource to incentivize optimal behavior:
$$\ell_r(|x|_r) + \tau_r(|x|_r).$$

## 12 Auctions

### Utility of a Bidder

$$\text{Utility Bidder } i = \begin{cases} 0, & \text{if loses auction,} \\ v_i - p, & \text{if wins auction,} \end{cases}$$
where $v_i$ is the bidder's valuation and $p$ is the price paid.

### Sealed-Bid Auction

- Each bidder privately submits a bid to the auctioneer.
- The auctioneer:
  1. Decides who gets the good (e.g., highest bid wins).
  2. Sets the selling price (affects bidder behavior).
- Bidders underbid because bidding true valuation yields zero utility. The degree of underbidding depends on unseen competitor bids.

## Second-Price Auction (Vickrey)

- In a second-price auction, every bidder has a dominant strategy: bid their true valuation.
- Key properties:
  1. Dominant-Strategy Compatible (DSIC): Bidding truthfully is always the best strategy.
  2. Surplus Maximization: Bidding truthfully maximizes the social surplus:
  $$\sum_{i=1}^{n} v_i x_i,$$
  where $x_i = 1$ if bidder $i$ wins.
  3. Efficient Implementation: Can be implemented in linear time.
- Truthful bidding avoids the following risks:
  1. Overbidding: May result in winning at a price greater than valuation, causing negative utility.
  2. Underbidding: May result in losing an auction even when the item's value exceeds the second-highest bid.
- Utility of a truth-telling bidder:
$$U_i = \begin{cases} v_i - B, & \text{if wins (where } B \text{ is the second-highest bid),} \\ 0, & \text{if loses.} \end{cases}$$

## Sponsored Search Auctions

- Goals:
  1. DSIC: Truthful bidding should be dominant and yield non-negative utility.
  2. Surplus Maximization: Truthful bidding should maximize social surplus.
  3. Polynomial-Time: Assignments and payments should be computable efficiently.
- Process:
  1. Assign slots to bidders in descending order of bids to maximize surplus.
  2. Set prices to ensure DSIC behavior.

## Myerson's Lemma

- Setting:
  1. Allocation Space $\mathcal{X}$:
     - Single-item auctions: $\mathcal{X} = \{0,1\}^n$, with $\sum_i x_i = 1$.
     - Sponsored search: $x_i = \alpha_j$ if bidder $i$ is assigned slot $j$.
  2. Allocation Rule $x(b) : \mathbb{R}^n_{\geq 0} \to \mathcal{X}$:
     - Single-item auctions: Allocate the good to the highest bid.
     - Sponsored search: Assign slot $j$ to the $j$-th highest bid.
  3. Payment Rule $p(b) : \mathbb{R}^n_{\geq 0} \to \mathbb{R}^n_{\geq 0}$:
     - Single-item auctions: Second-price payment.
     - Sponsored search: Determined via Myerson's Lemma.
  4. Utility:
     $$U_i(b) = v_i x_i(b) - p_i(b).$$
- Myerson's Lemma:
  1. An allocation rule $x$ is implementable if and only if it is monotone:
     $$b_i' \geq b_i \implies x_i(b_i', b_{-i}) \geq x_i(b_i, b_{-i}), \quad \forall b_i', b_i, b_{-i}, \forall i.$$
  2. For monotone $x$, there exists a unique payment rule $p$ such that $(x, p)$ is DSIC.
  3. The payment rule can be derived analytically.
- Observations:
  - Monotonicity ensures implementability.
  - The payment rule is unique.

## Proof of Monotonicity and Payment Rule

- DSIC implies that truthful reporting is the best strategy:
  $$z \cdot x(z) - p(z) \geq z \cdot x(y) - p(y),$$
  and
  $$y \cdot x(y) - p(y) \geq y \cdot x(z) - p(z).$$
- Combining, the "sandwich inequality" holds:
  $$z \cdot [x(y) - x(z)] \leq p(y) - p(z) \leq y \cdot [x(y) - x(z)].$$
- Implications:
  - $x(z)$ must be monotone; otherwise, the inequality is violated.
  - The payment difference $p(y) - p(z)$ is bounded by changes in allocation probability scaled by $z$ and $y$.

## Guessing the Price Function

- Use discontinuities in $x$ to derive $p$:

$$z \cdot (x(y) - x(z)) \leq \Delta p \leq y \cdot (x(y) - x(z)).$$

- At a discontinuity:

$$\Delta p = h \cdot z,$$

  where $h$ is the height of the discontinuity.

- Fix $p(0) = 0$. Candidate formula:

$$p_i(b_i, b_{-i}) = \sum_{j=1}^{l} z_j \cdot (\text{jump in } x_i(\cdot, b_{-i}) \text{ at } z_j).$$