15-382 Collective Intelligence

Salman Hajizada

First Edition

Disclaimer

This document aims to summarize the content of the slides for 15-382, including what the author considers important. As always, the definition of important is highly subjective so the author might have omitted something that was important to another person, or included something that is trivial to another.

Good luck,

SH

Dynamical Systems

Fingerprints of Complex Systems

- Multi-agent / multi-component
- Decentralized
- Local interactions
- Dynamic

Types of Abstract Models:

Agent-based Mechanistic implementation of the multi-component interactions.

Mathematical (white-box) Formally describe the relations among the relevant components.

Black-box Input-output pairs from the system are used to predict...

Statistical Describing patterns and correlations between variables.

Systems of ODEs

Here is a system of $n \geq 1$ Ordinary Differential Equations

$$\begin{cases} \frac{dx_1}{dt} = f_1(\mathbf{x}(t)) \\ \frac{dx_2}{dt} = f_2(\mathbf{x}(t)) \\ \vdots \\ \frac{dx_n}{dt} = f_n(\mathbf{x}(t)) \end{cases}$$

where $\mathbf{x}(t)$ is an n-dim vector.

A continuous-time Dynamical System is defined by a system of differential equations:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}, t; \theta) \quad \text{or} \quad \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t; \theta)$$

where $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ specifies how each component of the state evolves as t changes. It can depend on a set of given parameters θ

Some definitions:

- Initial conditions: where the system is at the beginning of the evolution: $\mathbf{x}(t_0)$
- Phase space: space of all possible states
- Trajectory: the curve traces by $\mathbf{x}(t)$ in the phase space starting from $\mathbf{x}(t_0)$

- Solution: is in the form $\mathbf{x}(t; t_0)$ that defines a family of time trajectories in the phase space. Once we fix t_0, we fix a unique trajectory

Vector fields and flows

How are solutions built? At any point, $\mathbf{f}$ assigns a vector that shows where the point is heading (direction of motion).

If we plot these arrows (vectors) in the phase space, we get can get an idea of how the system evolves.

Flow: $\Phi : \mathtt{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is the collection of all trajectories generated by all possible starting conditions.

$\Phi(t, \mathbf{x}_0) = \mathbf{x}(t; \mathbf{x}_0)$

A fundamental theorem guarantees that **two orbits corresponding to two different initial solutions never intersect with each other** , except at equilibrium

Basic Properties

An ODE is linear if

- $\mathbf{f}(\mathbf{x}) = A\mathbf{x}$ (Homogeneous)
- $\mathbf{f}(\mathbf{x}) = A\mathbf{x} + b$ (Affine)

Linear ODE enjoys closed form solutions, non-linear ODEs usually not

A system is autonomous if time doesn't appear in expression of $\mathbf{f}$.

Facts:

- Any n-order ODE can be rewritten as a system of 1st order ODEs in $\mathbb{R}^n$
- Any Non-Autonomous ODE can be rewritten as an autonomous one

So we will focus on 1st order, autonomous and linear ODEs

Solving!

General form of linear ODE:

$$\dot{\mathbf{x}} = A\mathbf{x}, \quad x \in \mathbb{R}^n$$

A solution is a function $\mathbf{x}(t)$ that satisfies the vector field A.

(Lots of derivation out is skipped, here's how to solve)

1. Solve $\det(A - \lambda I) = 0$ for λ
2. The roots λ_i are eigenvalues of A
3. For each λ_i, there exists a non-null eigenvector $\mathbf{u}_i$
4. Together they yield one solution: $\mathbf{x}(t) = \mathbf{u}_i e^{\lambda_i t}$
5. Each distinct eigen-pair gives ONE independent vector solution

6. The general solution is then the combination of these terms: $\mathbf{x}(t) = c_1 e^{\lambda_1 t}\mathbf{u}_1 + \cdots + c_n e^{\lambda_n t}\mathbf{u}_n$ (at most n terms)

Important: the above is strictly true only if all eigenvalues are distinct

Matrix Exponential representation: $\mathbf{x}(t) = e^{At}\mathbf{x}(0)$ where $\mathbf{x}(0)$ is a generic initial condition

Exponentials and Asymtotic Behavior

Since the solution is a sum of exponentials, stuff is being pulled in the direction of the eigenvectors, weighted by their corresponding signed eigenvalues.

If the real part of $\lambda_i > 0$, mode i is unstable/diverging.

If the real part of $\lambda_i < 0$, mode i is stable/contracting.

At each point, the solution **mixes** the modes.

Equilibrium points

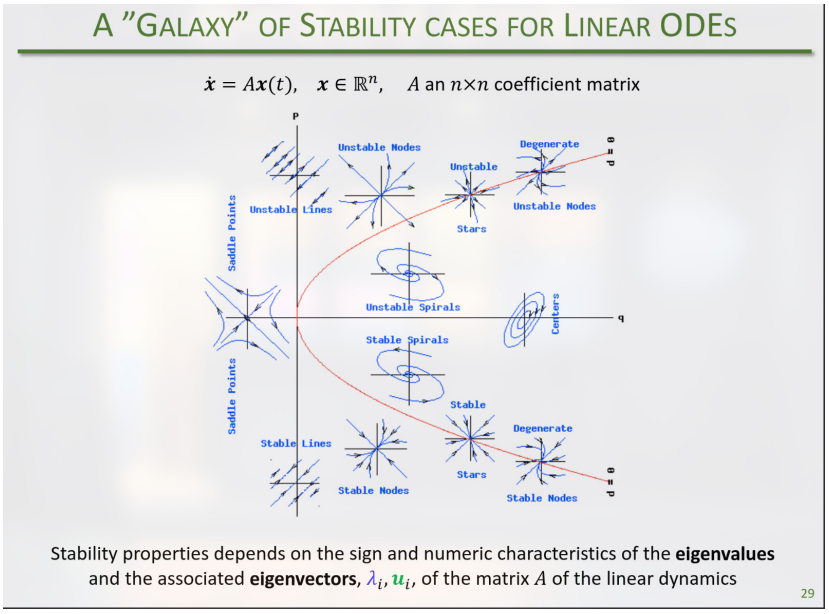
A state $\mathbf{x}_e$ is an equilibrium state of a system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ if when at a time t_0 the sytem is at $\mathbf{x}_e$ then it stays there FOREVER

Why? Velocity of the field in $\mathbf{x}_e$ is null: $\mathbf{f}(\mathbf{x}_e) = 0$

For a linear ODE, the equilibrium points are the points of the **Null Space** (solutions to $A\mathbf{x} = 0$) Theres one trivial solution at $\mathbf{x} = \mathbf{0}$ if A is invertible, o.w. inifnitely many solutions

Taxonomy of equilibria

Equilibrium Type	System's Behavior	Trajectories
Equilibrium state	If at or arrives at $\mathbf{x_e}$, it stays at $\mathbf{x_e}$	Trajectory is constant: $\mathbf{x}(t) = \mathbf{x_e}$
Stable equilibrium (Lyapunov)	If started close to $\mathbf{x_e}$, stays close to $\mathbf{x_e}$ forever	Nearby trajectories remain in a neighborhood of $\mathbf{x_e}$
Asymptotically stable equilibrium	If started close to $\mathbf{x_e}$, stays close to $\mathbf{x_e}$ and moves toward $\mathbf{x_e}$ as $t \to \infty$	Nearby trajectories converge to $\mathbf{x_e}$
Unstable equilibrium	Even if started very close to $\mathbf{x_e}$, eventually diverges from $\mathbf{x_e}$	Nearby trajectories diverge from $\mathbf{x_e}$



Linear System Classification by Eigenvalues

(for the saddle case, keep in mind a product is negative only if exaclty one of the numbers is negative)

Eigenvalues	Critical Point	Stability
$r_1, r_2 > 0$	Node (real, distinct)	Unstable
$r_1, r_2 < 0$	Node (real, distinct)	Asymptotically stable
$r_1 r_2 < 0$	Saddle	Unstable
$r_1 = r_2 \neq 0$	Node / Improper node	Same as sign of r_1
$r_{1,2} = \lambda \pm i\mu$	Spiral (focus)	Same as sign of λ
$r_{1,2} = \pm i\mu$	Center	Neutrally stable

Perturbations

For Pure Imaginary Eigenvalue, small perturbations add a tiny real part to the eigenvalues:

- $\lambda > 0$ (positive real part) $\implies$ trajectories spiral outward (unstable spiral).
- $\lambda < 0$ (negative real part) $\implies$ trajectories spiral inward (stable spiral).

For Repeated Real Eigenvalues

- If the eigenvectors are linearly independent, the system stays a node, but may change to a saddle if the signs differents.
- If the eigenvectors are linearly dependent (a degenerate node), a small perturbation will typically turn it into either a spiral (if eigenvalues become complex) or a node (if eigenvalues become distinct real numbers).

Linearization around a Critical Point

Consider a nonlinear system:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n$$

Let $\mathbf{x}_0$ be a critical point: $\mathbf{f}(\mathbf{x}_0) = 0$.

Step 1: Linear Approximation

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + J_{\mathbf{f}}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

Since $\mathbf{f}(\mathbf{x}_0) = 0$, the linearized system is

$$\dot{\mathbf{x}} \approx J_{\mathbf{f}}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0), \quad \mathbf{y} = \mathbf{x} - \mathbf{x}_0$$

Step 2: Jacobian Matrix

$$J_{\mathbf{f}}(\mathbf{x}_0) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}_{\mathbf{x}=\mathbf{x}_0}$$

Step 3: Eigenvalues, Eigenvectors, and General Solution

Compute eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of $J_{\mathbf{f}}(\mathbf{x}_0)$. The general solution of the linearized system is

$$\mathbf{y}(t) = \sum_{i=1}^{n} c_i e^{\lambda_i t} \mathbf{v}_i, \quad \mathbf{x}(t) = \mathbf{x}_0 + \mathbf{y}(t)$$

Step 4: Stability Analysis

Stability is determined by the eigenvalues of the Jacobian $J_{\mathbf{f}}(\mathbf{x}_0)$, just like in the linear case.

Global Behavior and Nullclines

- **Basin of Attraction:** The set of all initial conditions that eventually lead a trajectory to the same stable equilibrium point.
- **Separatrix:** The boundary between different basins of attraction.
- **Isoline:** The set of points where a function takes the same value.
- **Isocline:** The set of points where the vector field has the same **slope**.
- **Nullcline:** A specific type of isocline. It's the set of points where the slope is either zero or infinite (i.e., where one component of the vector field is zero).

Nullcline fun facts:

1. The x**-nullcline** is the curve where $\dot{x} = f(x, y) = 0$.
2. The y**-nullcline** is the curve where $\dot{y} = g(x, y) = 0$.
3. On the x-nullcline, the vector field has no horizontal component, so vectors can only point vertically (up or down).
4. On the y-nullcline, the vector field has no vertical component, so vectors can only point horizontally (left or right).
5. **Equilibrium points** are exactly at the intersection of the x-nullclines and y-nullclines (since $\dot{x} = 0$ and $\dot{y} = 0$ simultaneously).
6. Nullclines divide the phase space into regions. In each region, the sign of $\dot{x}$ and $\dot{y}$ is constant.

Limit cycles

A periodic orbit is just any trajectory that forms a closed loop.

A limit cycle is an isolated closed trajectory. This means neighboring trajectories are not closed; they either spiral into the limit cycle (a stable limit cycle) or spiral away from it (an unstable limit cycle). There's also mixed scenarios (half-stable)

Example: Van der Pol Oscillator

2D is Boring

Theorem 1. Any closed trajectory muyst encolse at least one equilibrium

Theorem 2. Poincare-Bendixson Theorem:

IF a trajectory is trapped in a closed, bounded region R,

AND this region R contains no equilibrium points,

THEN the trajectory must eventually approach a limit cycle.

In other words, 2D systems cannot have chaos

Chaos and Strange Attractors

3 main types of attractors: Fixed Points, Limit Cycles and Strange Attractors (the last appears only in 3D+)

Properties of strange attractors:

- Sensitive dependence on initial conditions: two initial conditions very close to each other become very far apart as time goes on (but remain confined in the set that defines the attractor)
- Fractal dimensions (e.g. 2.06). Non-integer

Example: Lorenz Attractor

1. For a parameter r=21, trajectories spiral into one of two stable fixed points.
2. For r=28, the fixed points become unstable. Trajectories are still bounded, but they never settle down. They move from one "wing" of the attractor to the other in an aperiodic, unpredictable way.

Definition 1. Chaos: **aperiodic long-term behavior** in a **deterministic** system that exhibits **sensitive dependence on initial conditions**

Iterated Maps

Intro

Discrete-time dynamical systems have a state which is only defined at integer steps.

The state is:

$$\mathbf{x}(n) = (x_1(n), \ldots, x_k(n))$$

where k is the dimension of the system and n is an integer step parameter

and the rule is:

$$\mathbf{x}(n) = (f)((\mathbf{x}(n-1), \ldots, \mathbf{x}(n-m)))$$

(so the system here depends on m previous states)

Can also be written nicely as $\mathbf{x}_n = \mathbf{f}(\mathbf{x}_{n-1}, \ldots, \mathbf{x}_{n-m})$

Next state is obtained by directly apply the map $\mathbf{f}$

1D Iterated Maps and Cobweb Plots

We focus on the simplest case: a 1D map $x_n = f(x_{n-1})$

- **Orbit:** sequence of points generated starting at x_0 and keep applying the map
- **Fixed point:** Point that maps to itself: $f(x^*) = x^*$

Sawtooth diagram: Plotting the $(x, f(x))$ diagram. No time shown, all it tells you is if the system is at x now, the next state will be at $f(x)$

Cobweb Plot construction algorithm:

1. Draw $y = f(x)$ and $y = x$
2. Start at initial point x_0 on horizontal axis
3. Move **vertically** to $y = f(x)$ (this is point x_0, x_1)
4. Move **horizontally** to $y = x$ (this is point x_1, x_1)
5. Move **vertically** to $y = f(x)$ again (this is point x_1, x_2)
6. Keep going lil bro

The intersections of $y = f(x)$ and $y = x$ are fixed points. We can analyze stability of a fixed point by looking at the plot.

Stability of fixed points

Some simple derivation because its interesting:

Consider a point near a fixed point, $x_n = x^* + \epsilon_n$. Next point $x_{n+1} = f(x^* + \epsilon_n) = f(x^*) + f'(x^*)\epsilon_n + O(\epsilon_n^2)$

So a linear approximation shows that $\epsilon_{n+1} = f'(x^*)\epsilon_n$. Let $\lambda = f'(x^*)$

Solving: $\epsilon_n = \lambda^n \epsilon_0$. If you know some basics you can infer stability from this alone but I will write a table.

$\lambda = f'(x^*)$	**Stability Type**	**Behavior Near** x^*
Stable (Attracting)		
$0 < \lambda < 1$	Monotonic	Converges from one side
$-1 < \lambda < 0$	Oscillatory	Zig-zag convergence (alternating sides)
$\lambda = 0$	Superstable	Very fast convergence
Unstable (Repelling)		
$\lambda > 1$	Monotonic	Diverges from one side
$\lambda < -1$	Oscillatory	Alternating divergence
Marginal		
$\lvert\lambda\rvert = 1$	Neutral	Linearization inconclusive

Logistic map and chaos

Logistic Map: $x_{n+1} = rx_n(1 - x_n)$ where $x_n \in [0, 1]$ is population, $r \in [0, 4]$ is growth rate

- $1 < r < 3$: The population converges to a single, stable fixed point $x^* = 1 - 1/r$.
- $r = 3$: The fixed point becomes unstable.
- $3 < r < 3.449\ldots$: The system no longer settles to one point. It settles into a stable period-2 cycle, oscillating between two values. This is a bifurcation.
- $r > 3.449\ldots$: The 2-cycle becomes unstable and splits into a stable period-4 cycle. This continues, creating an 8-cycle, 16-cycle, etc., in a period-doubling cascade.
- $r > r^\infty \approx 3.5699\ldots$: The cascade finishes, and the system enters the chaotic regime. The orbit becomes aperiodic, never settling down and seemingly random.

This behavior can be summarized in an **orbit diagram**.

- The x-axis is the parameter r.
- The y-axis plots the long-term attractor points for that r.

A **bifurcation** is a qualitative change in the long-term behavior of a system with a smooth variation of a parameter

A bunch of math deriving Lyapunov Exponents, but here is the final result:

$$\lambda = \lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{n-1} \ln |f'(x_k)|$$

The approximation for λ can be constructed numerically, by iterating the map!

- $\lambda < 0$: For stable fixed points and cycles
- $\lambda > 0$: For chaotic attractors
- $\lambda = 0$: This is the marginal case, which occurs at bifurcation points.
- λ is the same for all points in the basin of attraction of an attractor

Cellular Automata (CA)

CA properties

A **Cellular Automaton** is a multi-dimensional discrete-time dynamical system that is defined by the principle of locality. properties:

Systems's state is n-dimensional $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$

Discrete time : updated at discrete time steps

State components arranged according to a given topology

Neighborhood defined based on topology : $N(x_i) = x_j : x_j$ is a neighbor of x_i

Neighborhood is the range for a cell to be influenced by other cells

1D Example: $N(x_i) = x_{i-1}, x_i, x_{i+1}$

2D Examples:

- Von Neumann: The cell and its 4 neighbors (up, down, left, right).
- Moore: The cell and its 8 surrounding neighbors (a 3x3 box).

Locality of updates : Each state component x_i evolves according to a rule that depends only on its own state and those of its neighbors in $N(x_i)$.

Local-state transition function, $F_i : S(N(x_i)) \to S_i$ where S_i s the set of values that state component x_i can take, and S is the state values from the cells in the neighborhood set.

Initial Conditions : state at $t = 0$

Lattices and Boundaries:

Infinite/adaptive lattice: The grid grows as the pattern propagates

Finite lattice

- Hard boundary: fixed, edge cells have a fixed state
- Hard boundary: reflective, leftmost (rightmost) cell only diffuse right (left)
- Soft boundary: periodic boundary conditions, edges wrap around

Updating Schedules:

- Synchronous Updating: At time t, all cells read the states of their neighbors. Then, they all compute their next state. Finally, all cells update their state simultaneously to begin step $t + 1$.
- Asynchronous Updating: Cells update one at a time or in random groups. The order matters. A cell updating later in the step will see the already updated states of its neighbors who updated earlier.

Some Math

We assume they are homogeneous (same lattice, same N, and same rule F for all cells) and use synchronous updating.

Combinatorics Let:

- $k = |S|$ is the number of states per cell
- M is the number of cells

- r is the range ($floor(|N(a)|/2)$)

Then:

- Number of possible state configs: k^M
- Number of possible neighborhood configurations: $k^{|N|}$
- Number of possible evolution functions (rules): $k^{\left(k^{|N|}\right)}$ Why? This is the number of possible functions mapping from the set of neighborhood configurations (domain size $k^{|N|}$) to the set of possible next states (codomain size k). For each of the $k^{|N|}$ possible inputs, there are k possible outputs.

Wolfram Code

For Wolfram's 1D CA: $S = \{0, 1\}$ (so $k = 2$) and the neighborhood is $r = 1$ (i.e., $N(x_i) = \{x_{i-1}, x_i, x_{i+1}\}$, so $|N| = 3$).

Using our formula, the number of rules $= k^{\left(k^{|N|}\right)} = 2^{\left(2^3\right)} = 2^8 = 256$.

Four classes of Behavior and Chaos

Class	Behavior	Information Dynamics	Lyapunov Exponent
1	Evolves to a simple, stable, homogeneous state (all 0s or all 1s).	Small changes die; information is lost.	$\lambda \leq 0$
2	Evolves to simple periodic structures (stripes, oscillators).	Small changes may persist locally but do not spread.	$\lambda = 0$
3	Evolves to chaotic, aperiodic patterns (e.g., Rule 30).	Small changes spread out and affect distant regions.	$\lambda > 0$
4	Creates complex, localized, moving structures. (e.g., Rule 110 is **Turing complete**)	Small changes may or may not spread; irregular but structured dynamics.	$\lambda > 0$ (tends to 0)

Game of Life

- 2D Lattice of identical cells
- Moore neighborhood
- 2 states: dead or alive
- The 4 Rules:

  Loneliness A live cell with < 2 live neighbors dies.

  Overcrowding A live cell with > 3 live neighbors dies.

  Survival A live cell with 2 or 3 live neighbors lives on.

  Reproduction A dead cell with exactly 3 live neighbors becomes alive.

- Garden of Eden: A pattern that can only exist as initial pattern. In other words, no parent could possibly produce the pattern.

Networks

Basic Graph Theory, Terminology, Properties

Graph: two sets (V, E) of vertices/nodes and edges/arcs/links

Can be directed (interaction flows one way) or undirected (interaction flows both ways)

For Undirected graphs:

- Node degree k_i: number of edges adjacent to node i
- Handshake theorem: let L be the number of edges. Then $L = \frac{1}{2}\sum_{i=1}^{N} k_i$
- Average degree: $\langle k \rangle = \frac{2L}{N}$

For Directed Graphs:

- In-degree: number of edges going into node
- Out-degree: number of edges going out of the node
- Total degree $k_i = k_i^{\text{in}} + k_i^{\text{out}}$
- Average degree (either in or out): $\frac{L}{N}$

Degree Distribution: $p_k = \frac{N_k}{N}$ where N_k is number of nodes with degree k

In real networks degree distribution is highly heterogeneous

Complete graph: every possible edge is there. $L = \frac{N(N-1)}{2}$

In real networks, the number of edges is way less than that.

- **Walk:** A sequence of vertices (and corresponding edges) such that consecutive vertices are adjacent. Vertices and edges may repeat. For example, $(1 \to 2 \to 1)$ is a valid walk.
- **Path:** A walk in which all vertices (and therefore all edges) are distinct. For example, $(1 \to 2 \to 3)$ is a path, but $(1 \to 2 \to 1)$ is not.
- **Length:** The number of edges in a walk or path, or the sum of their weights if the graph is weighted.
- **Distance:** The length of the shortest path between two vertices. In unweighted graphs, this equals the minimal number of hops; in weighted graphs, the minimal total weight.
- **Shortest Path:** A path whose length equals the distance between its endpoints.
- **Diameter:** The maximum distance between any two vertices in the graph; equivalently, the length of the longest shortest path.

Connected graph is an undirected graph where there exists a path between every pair of nodes

For directed graphs there is

- Strongly connected if for every pair there is a path
- Weakly connected if every pair there is a path **when you ignore edge directions**

If the largest component in a graph has a large fraction of the nodes, we call it the giant component

Two representations for graphs

1. Adjacency list: a mapping from each node to it's neighborhood
2. Adjacency matrix: $N \times N$ matrix A such that $A_{ij} = 1$ if link (i, j) exists, and 0 otherwise.

More about adjacency matrices

1. For undirected graphs they are symmetric
2. Number of walks between nodes i, j can be calculated as follows: $(A^l)_{ij}$ gives #walks of length l.

Some measures

Average path length:

$$h = \frac{1}{2E_{\max} \sum_{i,j \neq i} h_{ij}}$$

where h_ij is the distance from i to j and $E_{\max} = n(n-1)/2$

Clustering Coefficient

Let L_i be the number of links between neighbors of i. Then the clustering coefficient of node i is

$$c_i = \frac{2L_i}{k_i(k_i - 1)}$$

Global clustering coefficient: $C = \frac{1}{N} \sum_{i=1}^{N} c_i$

Properties of real networks

Real networks are:

- Scale-free (from p_k): they have hubs
- Small world (from h): average distance is very small
- Locally dense (from C): clustering is very high

We will try to have a process that can create a network with these properties

Model 1: Erdos-Renyi (ER)

I DO NOT CAREEEEE ABOUT THE DOTS ABOVE THEIR NAMES

Generative process: each of the $\binom{N}{2}$ possible edges, an edge is created with probability p **Degree Distribution:**

$$p_k = \binom{N-1}{k} p^k (1-p)^{N-1-k}$$

Mean degree is $\bar{k} = p(N-1)$, Variance is $(N-1)p(1-p)$

For large, sparse networks that simplifies to Poisson distribution:

$$p_k = e^{-\bar{k}} \frac{\bar{k}^k}{k!}$$

Clustering Coefficient: $E[C] = p = \frac{\bar{k}}{N-1}$, i.e. small

Average path length grows as: $O(\log N)$

Verdict:

1. Scale-free: NO, p_k is Poisson (should be power law)
2. Small world: YES, low h
3. Locally dense: NO, C is very low (should be high)

Model 2: Watts-Strogatz (WS) (Small World)

This model was made to fix the clustering problem.

Generative process:

1. Start with a regular ring lattice, where each node is connected to m nearest neighbors. Graph starts with high C and h
2. Go through each edge, and with prob p rewire one end of the edge to a randomly chosen node.

What happens for different p?

- $p = 0$: Just a regular lattice, high C and h
- $p = 1$: Just a random graph, low C and h
- $p = 0.01$: A few rewired links act as shortcuts, dropping h dramatically (to around $logN$), while keeping a high C

Verdict:

1. Scale-free: NO, p_k is peaked at around m
2. Small world: YES, low h
3. Locally dense: YES, C is high

We are almost there

Model 3: Barabasi-Albert (BA) Scale-Free Model

Generative Process

Two main new mechanisms:

1. **Growth:** The network is not a fixed size. It starts with a small "seed" network, and at each time step, a new node is added.
2. **Preferential Attachment:** New node connects to m existing nodes. The probability $\Pi(i)$ of it connecting to an existing node i is not uniform. It is proportional to that node's current degree k_i:

$$\Pi(i) = \frac{k_i}{\sum_j k_j}$$

Rich get richer, rich nodes have higher chance of getting new links

Verdict:

1. Scale-free: YES, naturally produces a power-law degree distribution (p_k around k^{-3})
2. Small world: YES, low h
3. Locally dense: KINDA, C is a typically lower than in real networks

Summary of Network Models

Model	Scale-Free?	Small World?	High Clustering?
Erdos-Renyi (ER)	No	Yes	No
Watts-Strogatz (WS)	No	Yes	Yes
Barabasi-Albert (BA)	Yes	Yes	Kinda

Importance of a node: Network Centrality Measures

Certain positions within the network give nodes more power or importance. We will explore different types

Classic (Non-Recursive) Centrality Measures

Degree Centrality

The number of other nodes n is connected to.

Meaning: A node with high degree centrality has high potential communication activity

Betweenness Centrality

Number of shortest paths connecting all pairs of other nodes that pass through n

Meaning: A node with high betweenness centrality acts as a "bridge" or "gatekeeper" and has significant control over the flow of information.

Closeness Centrality

Measures how "close" a node is to all other nodes in the network.

$$C(n) = \frac{N}{\sum_m^{d(m,n)}}$$

where $d(m, n)$ is distance between m and n

Meaning: Efficiency of information spread. It answers the question, "How fast can I reach everyone else from this node?"

Self-Consistent (Recursive) Centrality

A node's importance is determined by the importance of its neighbors. This creates a recursive, self-referential definition.

Eigenvector centrality

The centrality of a node i is the scaled sum of centralities of its neighbors

$$c_i = \frac{1}{\lambda} \sum_{j \in N(i)}^{c_j}$$

Some math leads it to $A\mathbf{c} = \lambda\mathbf{c}$ where $\mathbf{c}$ is the principal eigenvector of the adjacency matrix

Computing eigvenector centrality

For a network with billions of nodes, we can't just "solve" this equation. We must compute it iteratively.

1. Power Method (Power Iteration)

This is the standard centralized algorithm. It uses iterative matrix multiplication to converge to the principal eigenvector.

1. **Initialize:** Choose an arbitrary nonzero vector $\mathbf{c}^{(0)}$ (e.g., all ones).
2. **Iterate:** Multiply by the adjacency matrix:
$$\mathbf{c}^{(t+1)} = A\mathbf{c}^{(t)}.$$
3. **Normalize:** To prevent numerical overflow, normalize at each step:
$$\mathbf{c}^{(t+1)} = \frac{A\mathbf{c}^{(t)}}{\|A\mathbf{c}^{(t)}\|}.$$
4. **Converge:** As $t \to \infty$, $\mathbf{c}^{(t)}$ converges to the principal eigenvector $\mathbf{c}_1$.

2. Gossip Algorithms (Decentralized Power Method)

Gossip algorithms implement the same idea in a distributed and asynchronous way, without a central coordinator.

- Each node i maintains its own estimate $c_i^{(t)}$.
- Nodes periodically *gossip* (push or pull) their current c_i values to their neighbors.
- Each node updates asynchronously:
$$c_i^{(t+1)} = \sum_{j \in N(i)} c_j^{(t)}.$$
Over time, these local updates collectively approximate the global power iteration, converging to the same principal eigenvector.

Alpha-Centrality

The idea is that the centrality is a combination of two things:

1. Recursive influence from its neighbors (scaled by α).
2. An "exogenous" or baseline importance e.

The defining equation is:
$$\mathbf{c} = \alpha A\mathbf{c} + \beta\mathbf{e}$$

Solving for $\mathbf{c}$:
$$\mathbf{c} = \beta(I - \alpha A)^{-1}\mathbf{e}$$

Since
$$(I - \alpha A)^{-1} = I + \alpha A + \alpha^2 A^2 + \ldots,$$
we can interpret each term as a contribution from paths of increasing length:

- $\beta I\mathbf{e}$ — baseline importance (paths of length 0),
- $\beta\alpha A\mathbf{e}$ — influence from direct neighbors (paths of length 1),
- $\beta\alpha^2 A^2\mathbf{e}$ — influence from neighbors-of-neighbors (paths of length 2),
- and so on for longer paths.

The parameter α controls the extent of influence propagation:

- For small α, only local structure matters (nearby nodes dominate).
- As α approaches $1/\lambda_1$ (where λ_1 is the largest eigenvalue of A), global structure dominates, and the measure converges to **eigenvector centrality**.

TODO (maybe): Add page rank stuff, and theres more math things there which looks like it wont come up... :)

Swarm Intelligence

Swarm Intelligence is the study and design of Complex Systems/ Multi-Agent Systems that

1. Potentically large number of locally interacting, decentralized, and distributed components, a swarm
2. Each component has a purpose that contributes to the peformance of the whole
3. Under some condiitons, the system displays emergent forms of collective / swarm intelligence

Communication Paradigms

The method of communication defines the type of SI system:

- Point-to-Point: Direct contact (e.g., ant antennation).
- Limited-Range Broadcast: A signal propagates locally (e.g., fish lateral lines detecting waves, visual cues).
- Indirect (Stigmergy): An agent modifies the environment, and another agent responds to that change later.
- Physical Mobility: Agents move through space, and their "network" is their set of current neighbors, which changes constantly.
- Static positioning, state evolution: connection topology and/or positioning in the environment do not change over time.

Particle Swarm Optimization (PSO)

Multi-agent black-box optimization framework inspired by social and roosting behavior of flocking birds

Background: Boids!

Realistic model of coordinated animal motion, agents follow three simple rules:

1. Separation: steer to avoid crowding local flockmates
2. Alignment: steer towards the average heading of local flockmates
3. Cohesion: steer to move toward the average position of local flockmates

Later, a **roost** was added. An attraction point in a simplified Boids-like simulation, such that each agent

1. Is attracted to the location of the roost
2. Remembers where it was closer to it
3. Tells its gang about its closest location to the roost

Eventually, (almost) all agents will land on the roost.

What if the roost is the unknown global optimum (min/max) of a mathematical function? And the "distance to the roost" is simply the quality of the function's value at that point?

Black-box optimization with PSO

We do not know the formula for $f(x)$. All we can do is "query" the function: provide an input x and observe the output $f(x)$. We must find the optimum by intelligently sampling the search space.

PSO is a derivative-free, black-box optimization algorithm.

Core Mechanic (In Words)

Swarm of particles are made, each particle i represents a potential solution.

- Position $\mathbf{x}_i \in \mathbb{R}^n$: Candidate solution of particle
- Velocity $\mathbf{v}_i \in \mathbb{R}^n$: Current direction and speed of particle

Each particle also has a memory:

- pbest, position with the best $f(x)$ particle i ever visited
- lbest, position with the best $f(x)$ any particle in ith social neighborhood ever visited (will be used later)

- gbest, position with the best $f(x)$ any particle ever visited

At each time step, the particle updates its velocity and position based on 3 influences:

1. Inertia component: keep going where its going already
2. Personal component: pull towards pbest
3. Social Component: pull towards gbest

Core Mechanic (In Math) :)

Velocity Update:

$$\mathbf{v}_i(t+1) = \omega \mathbf{v}_i(t) + c_1 r_1 (pbest_i - \mathbf{x}_i(t)) + c_2 r_2 (gbest_i - \mathbf{x}_i(t))$$

where

- ω is inertia weight (influence of old velocity)
- c_1, c_2 are acceleration coefficients (weighting the "pull" of the personal and social components).
- r_1, r_2 are random numbers, adding stochasticity to the search.

Position Update: Just move.

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1)$$

Parameters and Variants

What if only one agent? If left alone, each individual agent would behave like a stochastic hill-climber when moving in the direction of a local optimum, and then it will have a quite hard time to escape it.

Neighborhood Topology

There's no clear way tp know which topology is best.

- gbest (Global Best): Leads to fast convergence but can get trapped in local optima.
- lbest (Local Best): Each particle is pulled toward the best solution found by its immediate topological neighbors. Slower but explores the space more effectively, making it better for complex, multi-modal problems.

Acceleration Coefficients

The balance between c_1, c_2 two defines the swarm's search strategy:

- $c_1 > 0, c_2 = 0$: (Independent). No social influence, set of independent hill climbers
- $c_1 = 0, c_2 > 0$: (Social-only). No personal memory, entire swarm is pulled only towards single best-known point. One stochastic hill-climber
- $c_1 = c_2 > 0$: Attracted to average of pbest, gbest
- $c_2 > c_1$: (More Social). Better for unimodal problems
- $c_1 > c_2$ (More Personal). Better for multimodal problems

Inertia Coefficient

The weight was added to control balance between exploration and exploitation:

- $\omega \geq 1$: velocities increase over time, swarm diverges. Exploration.
- $0 < \omega < 1$: particles decelerate, convergence depends on c_1, c_2. Explotation.

Constriction Coefficient: idk what this is, TODO understand it.

Fully Informed PSO (FIPS)
Like **lbest**, FIPS uses neighborhood information, but more democratically. Instead of moving toward the single best particle, a particle is attracted to a weighted average of all neighbors' personal bests. This slows convergence slightly but reduces the chance of getting stuck in local optima.

Binary / Discrete PSO
PSO can be adapted for binary search spaces ($\mathbf{x}_i \in \{0,1\}^n$):

- Velocity $\mathbf{v}_i$ is continuous and updated normally.
- Each component v_{ij} is interpreted as the probability of a 1.
- Probability is computed via a sigmoid: $s(v_{ij}) = 1/(1+e^{-v_{ij}})$.
- Positions are updated stochastically:

$$x_{ij}(t+1) = \begin{cases} 1 & \text{if rand() ¡ } s(v_{ij}(t+1)) \\ 0 & \text{otherwise} \end{cases}$$

Optimization

Comparing PSO with single optimizing agents. We saw for PSO if left alone, each agent would behave like a memory-based random searcher, biased toward its own best-so-far.

A conservative blind explorer, with no reliable signal to use for improving.

Hill-Climber Optimizing Agent

Smarter Optimizing Agent. Like climbing Everest in thick fog with amnesia.

- Start at a random point
- Sample the neighborhood of the current point
- Always moves in direction of local imporovement
- Repeat until no neighbor is better (it has reached a peak).

Variants to improve it:

1. Stochastic hill-climber: introduces randomness in the selection of neighbor points
2. Random Restarts: escape mechanism for local optima

Hill-climbing is powerful for Constraint Satisfaction Problems (CSPs), like n-Queens. It can solve large instances of n-queens ($n = 106$) in a few seconds

Gradient-Based Agents

The Gradient ($\nabla f(\mathbf{x})$)
The gradient is the vector of partial derivatives

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right),$$

and it points in the direction of the steepest increase of the function f at $\mathbf{x}$.

Gradient Descent Algorithm
To minimize a function, an agent iteratively moves in the direction of the negative gradient:

1. Start at an initial point $\mathbf{x}_0$.
2. Compute the descent direction: $\mathbf{d}_k = -\nabla f(\mathbf{x}_k)$.
3. Choose a step size α_k (how far to move).
4. Update the position: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.

5. Repeat until the gradient is approximately zero ($\nabla f(\mathbf{x}) \approx 0$), indicating a minimum.

For convex functions we are guaranteed to get to the local and global minimum in a finite (possibly small) number of iterations

For non-convex functions, the final local optimum depends on where we start from

Applications for ML

Optimization function represents the classification Error of a ML model during training, i.e., the total Loss on on training set

$$\sum_{i=1}^{m} l(\hat{y}^i(\mathbf{w}^i, \mathbf{x}^i), y^i)$$

Gradient Descent Procedure for OLSR

For a linear model with quadratic loss:

$$E(\mathbf{w}; X, Y) = \frac{1}{2}\sum_{i=1}^{N}(\mathbf{w}^\top x_i - y_i)^2$$

The gradient is:

$$\nabla E(\mathbf{w}) = \sum_{i=1}^{N}(\mathbf{w}^\top x_i - y_i)x_i$$

Gradient descent update:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\sum_{i=1}^{N}(\mathbf{w}^{(t)\top} x_i - y_i)x_i$$

1. Initialize $\mathbf{w}^{(0)}$ (e.g., random or zero vector), $t \leftarrow 0$.
2. Repeat until convergence or maximum iterations N:
   a) Compute gradient: ∇E
   b) Choose step size α (fixed or adaptive)
   c) Update parameters: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\nabla E$
   d) If $\|\mathbf{w}^{(t+1)} - \mathbf{w}^{(t)}\| <$ tolerance, break
3. Return $\mathbf{w}^{(t+1)}$ as the solution.

Batch vs Stochastic GD

- Batch Mode (Regular): Gradient at each step is computed over the entire function, This is accurate but impossibly slow if $m = 1$ billion.
- Stochastic Gradient Descent: Gradient at each step is computed over one single component / data point selected out of the m ones. Takes many more steps (but each step is computationally light). The steps are very fast but "erratic".
- Mini-batch (stochastic) Gradient Descent: Gradient at each step is computed over one subset of data points. This is good balance of speed and stability.

Momentum and Adam

SGD can be slow or get stuck

Momentum (MGD)
Adds an inertia term by combining the current gradient with the previous update, allowing the agent to roll through small bumps and accelerate along gentle slopes:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \alpha \nabla f(\mathbf{w}_t), \quad \mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{v}_t$$

Adam (Adaptive Moment Estimation)
A popular optimizer combining two ideas:

- **Momentum ($\mathbf{m}_t$)**: exponentially weighted average of past gradients (1st moment) to determine direction.
- **Adaptive scaling ($\mathbf{v}_t$)**: exponentially weighted average of past squared gradients (2nd moment) to adapt learning rates per parameter.

Adam updates each weight using $\mathbf{m}_t$ for direction and $\mathbf{v}_t$ for adaptive step sizes.

Comparing ADAM against PSO

One core difference: ADAM is a Local Optimizer, PSO is a Global optimizer!

In general Adam worse, unless the shape is nice (Adam did well on ellispoid, got smoked in griewank, rastrigin, schwefel)

Hybrid?

If PSO is a great "explorer" and Adam is a great "exploiter," the obvious solution is to combine them.

The Strategy:

- PSO Phase: Let the PSO swarm run for several iterations. The particles spread out and find the most promising valleys.
- Local Search Phase: Take the best particles and run a local optimizer starting from their positions.
- Cooperation: The local optimizer will find the exact bottom of that valley. This new accurate position is then fed back to the swarm as the new gbest.

Local Search

Local Search is a fundamental optimization framework

Core Concept

They are iterative algorithms that each step consider a single current state, and try to improve it by moving to one of its neighbor state that has a better evaluation score

- Search for an improvement is done in local neighborhood only
- Hill climbing, gradient are specific instances of LS

Local Search methods work with complete problem states/solutions, s, i.e., all necessary variables are assigned

Neighborhoods in discrete domains

- 1-flip N for 0-1 vectors
- 2-swap N for permutation vectors
- k-exchange N, $N(s)$ of a state s is the set of states s' that differ from s up to k solution components/values

A small neighborhood is fast to search at each step, but the search is less powerful.

A large neighborhood is slower to search but can find better solutions.

Generic algorithm

Procedure: LocalSearch_SearchByIterativeSolutionModification(π)

Inputs:
- π : optimization problem instance (class II)
- Σ : set of feasible solutions for π
- $\mathcal{N}$: neighborhood structure (may depend on (s, t, m))

Functions:
- eval(s) : evaluation function
- terminate($s, \pi, \mathcal{N}, t, m$) : stopping condition
- get_next($\mathcal{N}, \pi, m$, eval) : proposes a neighbor and updates memory
- accept(s', s, t, m) : acceptance criterion
- update_best(π, s, t, m) : tracks best solution found

Initialization:
- $t \leftarrow 0$
- $m_t \leftarrow \emptyset$ (*memory, e.g. tabu list, stats, caches*)
- $s_t \leftarrow$ InitialFeasibleSolution(π, Σ)

Main Loop:
- **while** not terminate($s_t, \pi, \mathcal{N}_t, t, m_t$):
  - $(s', m_t) \leftarrow$ get_next($\mathcal{N}_t(s_t, \pi), \pi, m_t$, eval)
  - **if** accept(s', s_t, t, m_t):
    - $s_{t+1} \leftarrow s'$
    - $m_t \leftarrow$ update_best(π, s_{t+1}, t, m_t)
  - $t \leftarrow t + 1$

Return:
- **if** AtLeastOneFeasibleSolutionGenerated(m_t, Σ):
  - **return** BestSolutionFound(m_t)
- **else:**
  - **return** "No feasible solution found!"

Local Search for TSP: Neighborhood Structures

Local Search for the TSP relies on defining a neighborhood

2-opt

1. Select two non-adjacent edges, e.g., (A, B) and (C, D).
2. Remove them, creating two separate paths.
3. Reconnect the paths as (A, C) and (B, D).

Effect: Reverses the segment between the two edges, effectively "uncrossing" the tour.

Complexity: $O(N^2)$ possible edge pairs.

3-opt

1. Delete three edges, splitting the tour into three paths.
2. Reconnect them in one of $2^3 = 8$ possible configurations (including the original).

Effect: Allows both reversed and non-reversed segments, enabling solutions unreachable by 2-opt.

Complexity: $O(N^3)$ possible triples of edges.

Note: Necessary for Asymmetric TSP (ATSP) where $d(A, B) \neq d(B, A)$.

4-opt (Double Bridge)

Features:

- Does not revert tour segments (useful for ATSP).
- Achieves $O(N^2)$ complexity with clever implementation.
- Commonly combined with 2-opt for improved performance.

Solution Modification vs Construction

- Solution Modification (Local Search): You start with a complete solution (e.g., a full TSP tour) and make local modifications to iteratively improve it.
- Solution Construction: You start with an empty solution and incrementally build it, one piece at a time, until it is complete.

Procedure: Construction_Metaheuristic(π)

Inputs:
- π : optimization problem instance (class II)
- Σ : set of feasible (complete) solutions for π
- I : index set of decision variables (e.g., $I = \{0, 1, 2, \dots\}$)
- $\mathcal{X}$: domain sets of decision variables (e.g., $\mathcal{X} = [0, 1]$)
- eval(s) : evaluation function of assignments

Construction Operators (problem-specific):
- select_variable_index($I \mid s_t$)
- assign_variable_value($\mathcal{X} \mid s_t$)
- include_in_solution(s_t, i, x_i)
- update_cost($J_t; x_i \mid s_t$)

Termination Policy:
- termination_criterion(s_t, t)

Initialization:
- $t \leftarrow 0$
- $s_t \leftarrow \emptyset$ (*partial solution*)
- $J_t \leftarrow 0$ (*cost accumulator*)

Main Loop:
- **while** ($s_t \notin \Sigma$) and not termination_criterion(s_t, t):
  - $i_t \leftarrow$ select_variable_index($I \mid s_t$)
  - $x_{i_t} \leftarrow$ assign_variable_value($\mathcal{X} \mid s_t$)
  - $s_{t+1} \leftarrow$ include_in_solution(s_t, i_t, x_{i_t})
  - $J_{t+1} \leftarrow$ update_cost($J_t; x_{i_t} \mid s_t$)
  - $t \leftarrow t + 1$

Return:
- **if** ($s_t \in \Sigma$):
  - **return** (s_t, J_t)
- **else:**
  - **return** "No feasible solution found!"

Robots, or if you fancy: Cyber-Physical Multi-Agent Systems (CP MAS)

Systems of multiple interacting cyber-physical agents (robots, sensors) combining mechatronic structures, communication, and processing.

The near-future vision is of "Robots Everywhere"

Design Objectives of Robot Swarms

- **Scalability:** Performance should degrade gracefully as swarm size increases.
- **Robustness:** Tolerance to individual robot failures or environmental changes.
- **Flexibility / Adaptability:** Ability to handle different tasks or environments.
- **Distributed Control:** No single point of failure; decision-making is local.

Swarm Intelligence Design Approach

The primary design paradigm is bottom-up:

- Relatively simple individual controllers.
- Relatively complex interaction patterns.
- Relies mostly on locality of interactions and communications (decentralized and distributed).
- Leverages emergence and self-organization.

Downsides of Swarm Design

The complexity leads to challenges:

- Predictability and formal guarantees of swarm behavior can be challenging.
- Finite-time performance is often hard to characterize.
- Efficiency might be mediocre when heavily relying on full decentralization and self-organization.

CPMAS Taxonomy

Feature	**can be:**	**or can be:**
Members	Homogeneous (interchangeable units)	Heterogeneous (units with different skills/capabilities)
Coupling	Loosely coupled (agents operate independently; cooperation optional — e.g., speedup)	Tightly coupled (agents depend on each other; require coordination/cooperation)
Goals	Non-cooperative (agents maximize individual utility; equilibrium concepts apply)	Cooperative (agents maximize a global objective; social welfare / optimization concepts)
Control	Centralized control (single decision authority or planner)	Decentralized / distributed control (local decision-making; peer-to-peer coordination)
Coordination & Planning	Explicit (direct communication or sharing of plans among agents)	Implicit (coordination emerges through actions that influence others without direct communication)

Table 1: *Note:* The entries are *independent* options used to describe a system; real systems may combine any of these (e.g., a heterogeneous but loosely coupled group, or a homogeneous yet tightly coupled group).

Core issue:

Communications are fundamental but face issues:

- Global schemes won't scale for large swarms.
- Need for ad hoc networking in infrastructure-less environments (e.g., post-disaster).
- Challenges in deciding what to communicate, how frequently, and to whom.

AntHocNet:

An example of managing a Mobile Ad Hoc Network (MANET) using a hybrid Ant Colony Optimization (ACO) approach. It uses ant agents to set up full paths (reactive) and local information exchange to maintain and improve them (proactive).

Task Allocation

Definition

Task Allocation is the problem of deciding "who does what, where, when and how" in a multi-robot system.

It involves finding a mapping / allocation A: T $\rightarrow$ R, which assigns a set of n_t tasks (T) to a set of n_r robots (R).

The goal is to find an assignment that maximizes overall global system utility.

MRTA Taxonomy

Dimension	Type 1	Type 2	Description
Robot Tasking (R)	Single-Task (ST)	Multi-Task (MT)	Can a single robot work on one task (ST) or multiple tasks simultaneously (MT)?
Task Type (T)	Single-Robot (SR)	Multi-Robot (MR)	Does a task require one robot (SR) or a coalition/team of robots (MR)?
Assignment Time (A)	Instantaneous Assignment (IA)	Time-Extended Allocation (TA)	Is the assignment instantaneous or one-shot (IA), or does it involve planning a schedule or route over time (TA)?

Table 2: Dimensions of multi-robot task allocation (MRTA) taxonomy.

Optimization Models for MRTA Problems

The "Hello World": ST-SR-IA

The simplest MRTA problem is the ST-SR-IA problem, where each robot can only do one task at a time (ST), each task requires only one robot (SR), and the assignment is instantaneous (IA).

Linear Assignment Problem (LAP) Solution: The optimal assignment can be found in polynomial time using the Hungarian Algorithm with a complexity of O(n^3), (where n is the number of robots/tasks).

Scenarios	Quick Definition	Important Result / Insight
Adding Task Priorities	Tasks have priority weights w_t indicating importance, added to the objective function.	Weighted assignment maximizes total utility $\sum_r \sum_t w_t U_{rt} x_{rt}$.
$\lvert R\rvert \neq \lvert T\rvert$ **(Unequal sets)**	Number of robots and tasks differ. Use dummy robots/tasks to balance.	Two IA rounds preserve polynomial-time solvability without affecting optimal real assignments.
Iterated Assignment	Utilities or task info change over time $\rightarrow$ recompute or adaptively update assignments.	BLE (2001): 2-competitive ($\geq$50% of optimal); L-ALLIANCE (1998): learns best assignments iteratively.
Online Assignment	Tasks revealed one-by-one; robots may not be reassignable.	MURDOCH (2002): greedy, 3-competitive; best possible for online assignment without reassignment.

Other MRTA Scenarios

- **ST–SR–TA:** Find disjoint routes or schedules for multiple robots to visit all tasks, minimizing total travel cost. *Model/Complexity:* Multiple Traveling Salesperson Problem (mTSP), NP-Hard.
- **MT–SR–IA:** Assign multiple tasks to a single robot, constrained by robot capacity/resources, to maximize total utility. *Model/Complexity:* Generalized Assignment Problem (GAP), NP-Hard.

- **MT–SR–TA: VRP:** Vehicle Routing Problem (VRP). Find optimal routes for a fleet of vehicles to serve a set of customers/tasks, including constraints like limited capacity and time windows. *Model/Complexity:* VRP, NP-Hard.
- **ST–SR–TA + Dependencies:** ST–SR–TA with precedence constraints (e.g., Task A must be completed before Task B can begin). *Model/Complexity:* Complex Scheduling / Precedence-Constrained Scheduling, NP-Hard.
- **Traveling Salesman Problem (TSP):** Find the shortest single tour that visits a set of n tasks (cities) exactly once and returns to the origin (single-robot route planning). *Model/Complexity:* TSP, NP-Hard ($O(2^n)$).
- **Vehicle Routing Problems (General):** Generalization of mTSP, where vehicles must respect real-world constraints like capacity limits, time windows, and precedence. *Model/Complexity:* VRP, NP-Hard.

Orienteering Problems (OP)

Orienteering Problems involve selecting a subset of tasks/locations to visit while maximizing total collected scores/profit, subject to route and time constraints.

- **Classic Orienteering Problem (Single Vehicle):** Also called Selective TSP, Maximum Collection, or Bank Robber Problem. Maximize profit collected within a route duration constraint.
- **Profitable Tour Problem:** Maximize profit minus travel cost for a single vehicle.
- **Prize Collecting Traveling Salesman Problem (PCTSP):** Minimize travel cost while achieving a given route profit, for a single vehicle.
- **Team Orienteering Problem (Multiple Vehicles):** Also called Multiple Tour Maximum Collection Problem. Maximize total profit across multiple vehicles under route duration constraints.

Set-Based Formulations

- **Set Covering:** Select a subset of activities so that all requirements are covered at least once, minimizing cost. → Models MT–MR–IA (robots may appear in multiple coalitions).

$$\min \sum c_j x_j \quad s.t. \sum a_{ij} x_j \geq 1$$

- **Set Packing:** Select disjoint subsets to maximize profit — each task covered by *at most one* robot. → Models ST–MR–IA (exclusive task allocation).

$$\max \sum p_j x_j \quad s.t. \sum a_{ij} x_j \leq 1$$

- **Set Partitioning:** All tasks covered *exactly once* by disjoint subsets. → Base for routing/scheduling models (e.g., VRP).

$$\sum a_{ij} x_j = 1$$

Set Models and Coalition Formation in MRTA

- **Set Covering:** All tasks covered, robots may overlap → models **MT–MR–IA**.
- **Set Packing:** Maximize unique task coverage (no overlap) → models **ST–MR–IA**.
- **Set Partitioning:** All tasks done once, strict non-overlap → used in routing/scheduling (**VRP, TSP**).

Task allocation often combines:

- *Selection* — choosing which agents or coalitions perform which tasks (set-based).
- *Ordering* — sequencing or scheduling tasks for each agent (routing/scheduling).

SUMMARY, MAPPING MRTA TYPE TO OPTIMIZATION MODEL

MRTA Type	Description	Classical Model Equivalent
SR–ST–IA	Each robot does one task; all tasks known at once	**Assignment Problem**
SR–ST–TA	Robots plan multiple sequential tasks (known ahead)	**Generalized Assignment / Scheduling**
SR–MT–IA	Robots can do multiple parallel tasks; tasks covered once	**Set Covering / Partitioning**
SR–MT–TA	Multi-task, time-extended routing problem	**Vehicle Routing (VRP)**
MR–ST–IA	Tasks require multiple robots	**Set Covering / Coalition Formation**
MR–ST–TA	Multi-robot tasks with time scheduling	**Multi-Robot Scheduling**
MR–MT–IA	Multiple tasks and robots concurrently	**Set Covering with Capacities**
MR–MT–TA	Most general case — everything dynamic	**Full Multi-Robot Planning (NP-hard)**