Einführung in die KI

Niclas Kusenbach
LaTeX version: SCHOUTER

Table of Contents

Contents

1 Introduction

1.1 What is AI?

Literature:

- Empfohlenes Begleitbuch: Russel and Norvig, Artificial Intelligence: A Modern Approach, 4. Edition 2020.

1.1.1 Definitionen (Definitions)

1.1.2 Definitions

There is no easy, official definition for AI. Two classic definitions are:

- **John McCarthy (1971):** "The science and engineering of making intelligent machines, especially intelligent computer programs." AI does not have to confine itself to methods that are biologically observable.
- **Marvin Minsky (1969):** "The science of making machines do things that would require intelligence if done by men".

1.1.3 Categories of AI

AI definitions can be classified along two dimensions

1. Thought processes/reasoning vs. behavior/action
2. Success according to human standards vs. success according to an ideal concept of intelligence (rationality)

- **Systems that think like humans:**
  - Cognitive Science.
  - Builds on cognitive models validated by psychological experiments and neurological data.
- **Systems that act like humans:**
  - The **Turing Test**
- **Systems that think rationally:**
  - Focus on "Laws of Thoughts," correct argument processes.
- **Systems that act rationally:**
  - Focus on "doing the right thing" (**Rational Behavior**).
  - A rationally acting system maximizes the achievement of its goals based on the available information.
  - This is more general than rational thinking (as a provably correct action often does not exist) and more amenable to analysis.

1.1.4 General vs. Narrow AI

- **General (Strong) AI:** Can handle *any* intellectual task that a human can. This is a research goal.
- **Narrow (Weak) AI:** Is specified to deal with a *concrete* or a set of specified tasks. This is what we currently use primarily.

1.2 What is Intelligence?

1.2.1 The Turing Test

- **Question:** When does a system behave intelligently?

- **Assumption:** An entity is intelligent if it cannot be distinguished from another intelligent entity by observing its behavior.
- **Test:** A human interrogator interacts "blind" (e.g., via text) with two players (A and B), one of whom is a human and one a computer.
- **Goal:** If the interrogator cannot determine which player... is a computer... the computer is said to pass the test.
- **Relevance:** The test is still relevant, requires major components of AI (knowledge, reasoning, language, learning), but is hard/not reproducible and not amenable to mathematical analysis.

1.2.2 The Chinese Room Argument

- **Question:** Is intelligence the same as intelligent behavior?
- **Assumption:** Even if a machine behaves in an intelligent manner, it does not have to be intelligent at all (i.e., without understanding).
- **Thought Experiment:** A person who doesn't know Chinese is locked in a room. They receive Chinese notes (questions) and have a detailed instruction book telling them which Chinese symbols (answers) to output based on the input symbols, without understanding it at all.
- **Result:** From the outside, the room "understands" Chinese (it behaves intelligently), but the person inside understands nothing.
- **Follow-up Question:** Is a self-driving car intelligent?

1.3 Foundations, Taxonomy & Limits

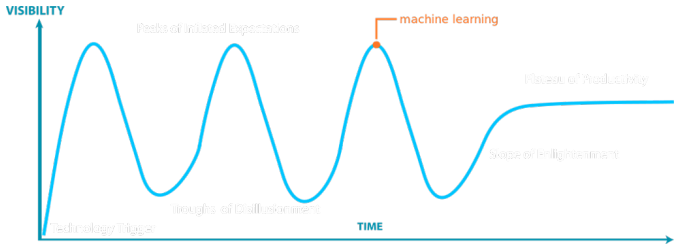
1.3.1 Foundations of AI

AI is an interdisciplinary field built on contributions from many areas:

- **Philosophy:** Logic, reasoning, rationality, mind as a physical system.
- **Mathematics:** Formal representation and proof, computation, probability.
- **Psychology:** adaptation, phenomena of perception and motor control.
- **Economics:** formal theory of rational decisions, game theory.
- **Linguistics:** knowledge representation, grammar.
- **Neuroscience:** physical substrate for mental activities.
- **Control theory:** ...optimal agent design.

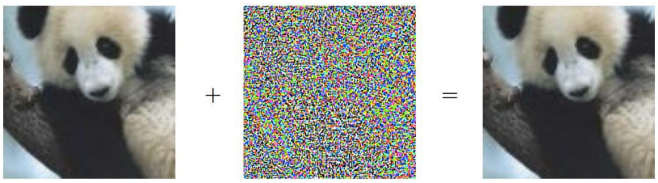
1.3.2 Taxonomy and History

- **Taxonomy: Artificial Intelligence** is the broadest field. **Machine Learning (ML)** is a subfield of AI. **Deep Learning** is a subfield of ML.
- **Subdisciplines of AI:** Include Machine Learning, Deep Learning, Search and Optimization, Robotics, Natural Language Processing (NLP), Computer Vision (CV), and Cognitive Science.
- **History:** The development of AI occurred in cycles, often called "AI Winters". Hype phases ("Peaks of Inflated Expectations") existed for "neural networks", "expert systems", and "machine learning".



1.3.3 Limits of Current AI

- **"A.I. is harder than you think":**
  - Current AI is often isolated to single problems.
  - AI models are **not without bias**.
  - There are **fundamental differences** in how AI perceives the world/environment.
- **AI can be tricked (Adversarial Examples):**
  - AI systems can be manipulated by perturbations (noise) often invisible to humans.
  - Example: An image of a "panda" is classified as a "gibbon" with high confidence after adding noise.



"panda"
57.7% confidence

"gibbon"
99.3 % confidence

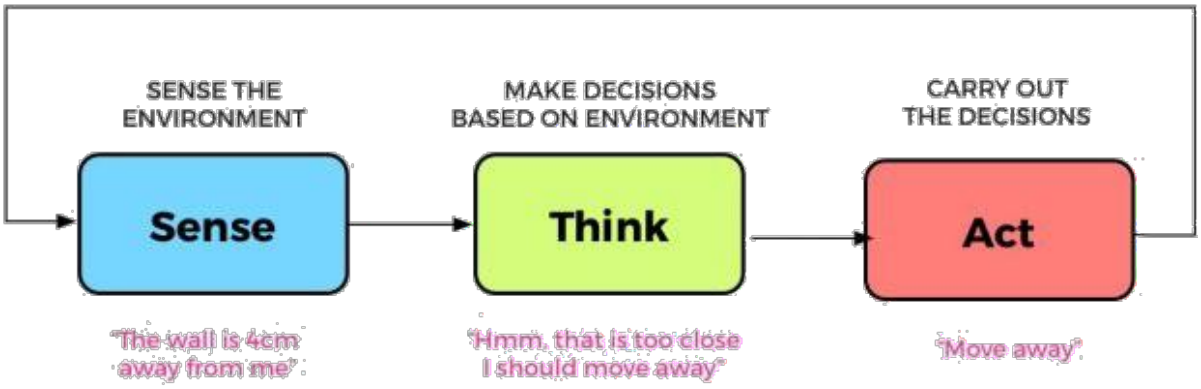
2 AI Systems: Agents and Environments

Definition: AI System

An AI system is defined as the study of (rational) **agents** and their **environments**. The system has two main parts:

1. **Agent:** Anything that can be viewed as perceiving its environment through **sensors** and acting upon that environment through **actuators**.
2. **Environment:** The surroundings or conditions in which the agent lives or operates. This can be real (e.g., streets for a self-driving car) or artificial (e.g., a chessboard).

The agent follows a continuous **Sense** → **Think** → **Act** loop.



2.1 Rationality

Rationality

- A **rational agent** is one that "does the right thing".
- A **rational action** is one that maximizes the agent's performance and yields the best positive outcome.
- **Key Point:** Rationality maximizes **expected** performance, not necessarily the *optimal* outcome. E.g., not playing the lottery is rational (positive expected outcome), even if playing could lead to the optimal outcome (winning).
- Rationality is **not** omniscient. An omniscient agent would know the *actual* outcome of its actions, which is impossible in reality.

A **performance measure** is a function that evaluates a sequence of actions.

General Rule for Design

Design the performance measure based on the **desired outcome**, not the desired agent behaviour.

2.2 Characteristics of Environments

The design of an agent heavily depends on the type of environment it operates in. Environments are characterized along several key dimensions.

Environment Dimensions

- **Discrete vs. Continuous:** Does the environment have a limited, countable number of distinct states (e.g., chess) or is it continuous (e.g., position and speed of a self-driving car)?
- **Observable vs. Partially/Unobservable:** Can the agent's sensors determine the *complete* state of the environment at each time point? If not, it is **partially observable** (e.g., a taxi cannot know pedestrian intentions, poker agent cannot see opponent's cards).
- **Static vs. Dynamic:** Does the environment change while the agent is acting/deliberating? A crossword puzzle is **static**; taxi driving is **dynamic** (other cars move).
- **Single Agent vs. Multiple Agents:** Is the agent operating by itself? Or does the environment contain other agents (e.g., other drivers, poker players)?
- **Accessible vs. Inaccessible:** Can the agent obtain *complete and accurate* information about the environment's state?
- **Deterministic vs. Non-deterministic (Stochastic):** Is the next state of the environment completely determined by the current state and the agent's action? Chess is **deterministic**. A self-driving car is **non-deterministic** (turning the wheel can have slightly different effects due to road friction, wind, etc.).
- **Episodic vs. Sequential:** In an **episodic** environment, the agent's experience is divided into "episodes". The quality of its action depends only on the current episode (perceive → act). In a **sequential** environment, the agent requires memory of past actions to make the best decision.

Key Distinction: Observable vs. Accessible

- **Accessibility** concerns the environment itself: whether the information exists and can *in principle* be obtained.
- **Observability** concerns the agent's *sensors*: whether they can actually perceive that information.

Environment	Discrete?	Observable?	Static?	Single Agent?	Accessible?	Deterministic?	Episodic?
Chess	Discrete	Observable	Static	Multi-Agent	Accessible	Deterministic	Sequential
Solitaire	Discrete	Observable	Static	Single Agent	Accessible	Deterministic	Sequential
Poker	Discrete	Partially Observable	Static	Multi-Agent	Partially Accessible	Stochastic	Sequential
Self-Driving	Continuous	Partially Observable	Dynamic	Single Agent	Inaccessible	Stochastic	Sequential
Medical Diagnosis	Discrete	Partially Observable	Static	Single Agent	Inaccessible	Stochastic	Episodic

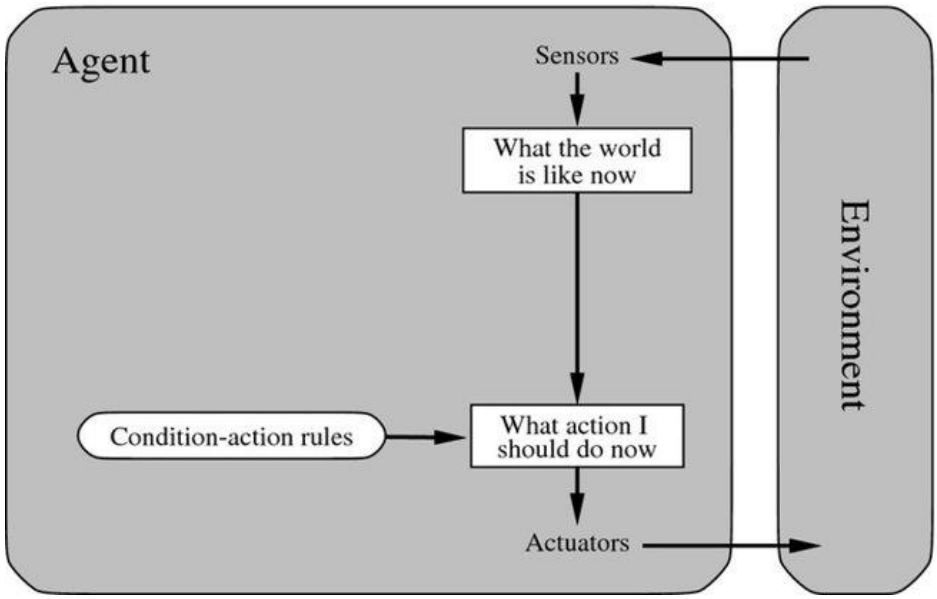
Table 1: *Characteristics of various environments*

2.3 Types of Agents

Agents are categorized based on their perceived intelligence and complexity.

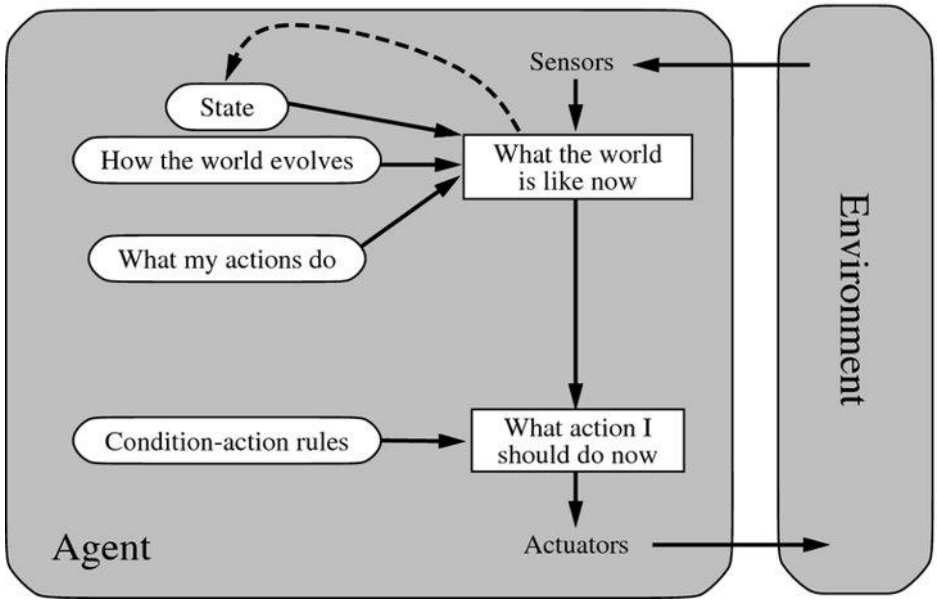
2.3.1 Reflex Agent

- Selects actions based **only on the current percept**, ignoring the percept history.
- Implemented with simple **condition-action rules**.
- Example: A thermostat (IF temp $< 20°C$ → turn on heater).
- **Problem:** Very limited. No knowledge of anything it cannot actively perceive.



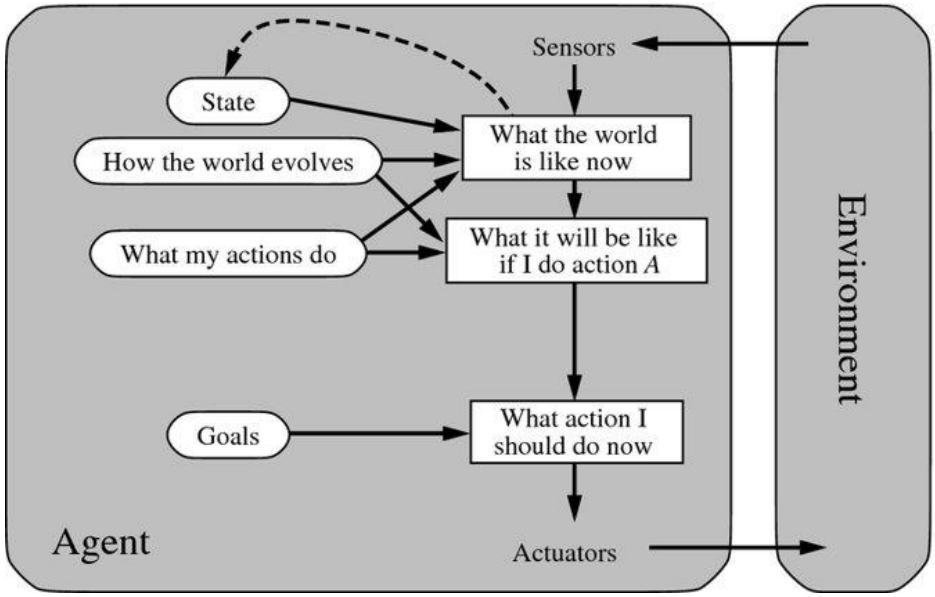
2.3.2 Model-based Agent

- These agents **keep track of the world state**.
- They maintain an **internal state (a world model)** that describes how the world evolves and how the agent's actions affect it.
- This allows the agent to handle partially observable environments.
- Example: A warehouse robot tracking inventory positions.



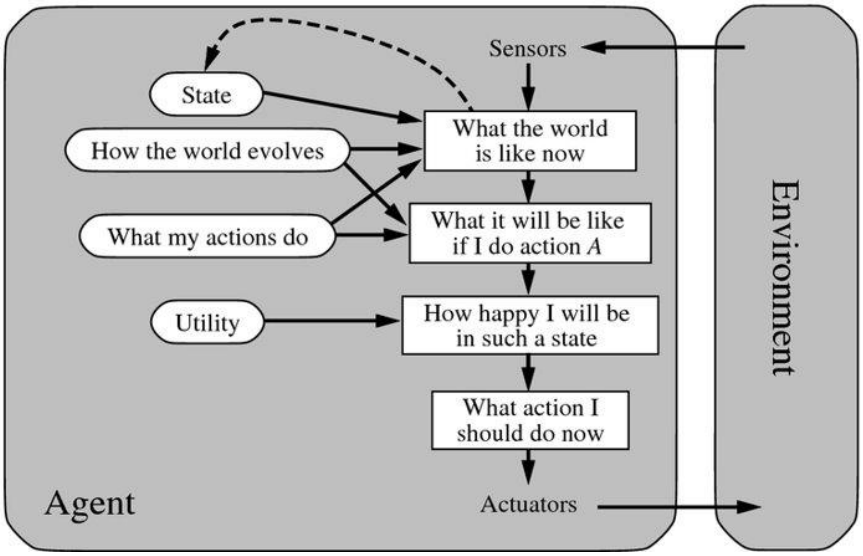
2.3.3 Goal-based Agent

- Builds on a model-based agent, but also knows what states are **desirable** (i.e., it has **goals**).
- This allows the agent to make decisions by considering the future, asking "What will happen if I do action A?" and "Will that action achieve my goal?".
- Example: A chess agent whose goal is to checkmate the opponent.



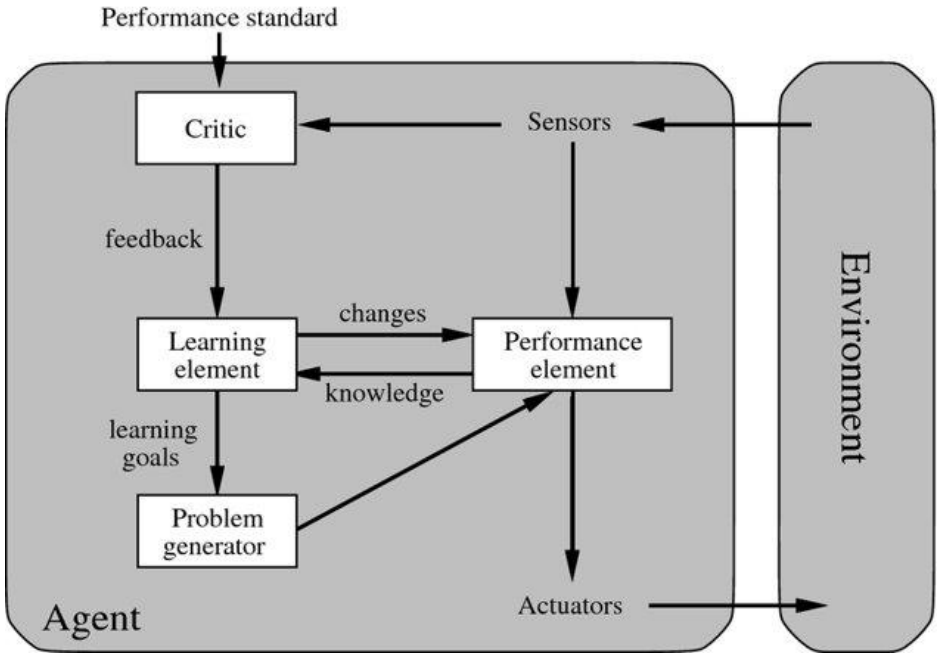
2.3.4 Utility-based Agent

- Goals provide a binary distinction (achieved / not-achieved). A **utility function** provides a continuous scale, rating each state based on the desired result ("how happy" the agent is).
- This is crucial for resolving **conflicting goals** (e.g., is speed or safety more important for a self-driving car?).
- Allows the agent to choose the action that maximizes its **expected utility**.



2.3.5 Learning Agent

- Employs a **learning element** to gradually improve and become more knowledgeable over time.
- Can learn from its past experiences and adapt automatically.
- More robust in unknown environments.



Four Components of a Learning Agent

1. **Learning Element:** Responsible for making improvements by learning from the environment.
2. **Critic:** Gives feedback on how well the agent is doing with respect to a fixed performance standard.
3. **Performance Element:** Responsible for selecting actions (this is the "agent" part).
4. **Problem Generator:** Responsible for suggesting actions that will lead to new (and potentially informative) experiences.

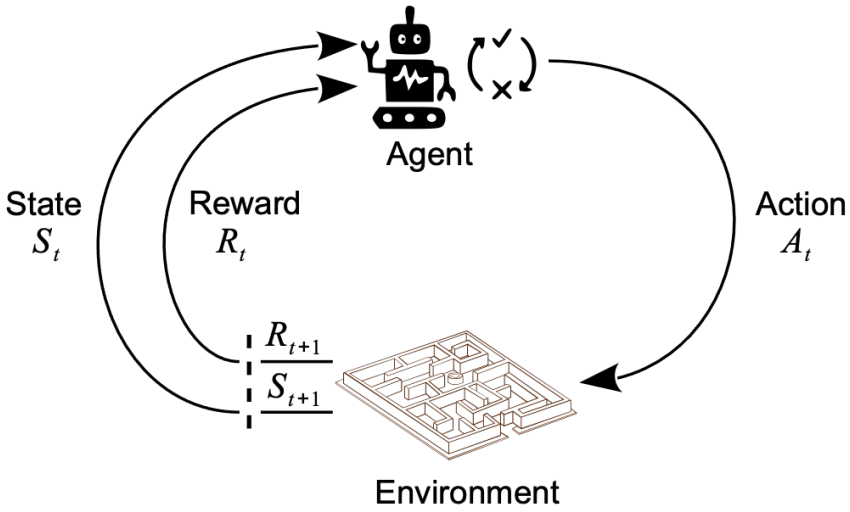
Agent Types Summary

- **Reflex agent:** reacts.
- **Model-based agent:** remembers.
- **Goal-based agent:** plans.
- **Utility-based agent:** optimizes.
- **Learning agent:** improves itself over time.

2.4 How to Make Agents Intelligent

There are several high-level approaches to selecting intelligent actions:

- **Search Algorithms:** Understand "finding a good action" as a search problem and use tree-based algorithms to find a solution (path to a goal).
- **Reinforcement Learning (RL):** Based on trial and error, similar to animal conditioning. The agent receives **rewards** (positive) or **pain/punishments** (negative) from the environment and learns to choose actions that maximize its cumulative reward.



- **Genetic Algorithms (GAs):** Inspired by Darwinian evolution ("survival of the fittest"). A **population** of agents is generated, evaluated by a **performance function**, and the best ones are "bred" (using **crossover** and **mutation**) to create a new, potentially better, generation.

3 Uninformed and Informed Search

3.1 Problem Formulation

Problem-solving agents are result-driven. They always focus on satisfying their goals, i.e., solving the problem. While problems are often given in a human-understandable way, we need to reformulate the problem for our agent. These agents employ algorithms to find solutions.

Steps to formulate a solvable problem:

1. **Formulate the goal**
2. **Formulate the problem** given the goal

3.1.1 Key Terminology

The State Space / States

A state describes a possible situation in our environment. The **state space** is a set of all possible situations (states).

Transition / Action

Transitions describe possible actions to take between one state and another. We only count direct transitions between two states (single actions).

Costs

Often transitions aren't alike and differ. We express this by adding a "cost" to each action. Often the goal in search algorithms is to **minimize the cost** to reach the goal.

A **single state problem** is defined by 4 items:

1. **State space and Initial state** Description of all possible states and the initial environment as state.
2. **Description of actions** Typically a function that maps a state to a set of possible actions in this state.
3. **Goal test** Typically a function to test if the current state fulfills the goal definition.
4. **Costs** A cost function that maps actions to its cost. An easy way is to have additive costs (sum of costs for all actions taken).

3.1.2 The State-Space Graph

The state space is the set of all states reachable from the initial state. It is implicitly defined by the initial state and the successor function, forming a **state-space graph**.

- **Path:** A sequence of states connected by a sequence of actions.
- **Solution:** A path that leads from the initial state to a goal state.
- **Optimal Solution:** A solution with the minimum path cost.

3.1.3 Core Search Definitions

Planning Problem

A planning problem is one in which we have an initial state and want to transform it into a desired goal, considering future actions and their outcomes.

Search

The process of finding the (optimal) solution for such a problem in the form of a sequence of actions.

3.2 Search Fundamentals

3.2.1 Tree Search vs. Graph Search

The state-space graph can be explored by building a **search tree**.

- **Tree Search:** Treats the state space as a tree. It does not keep track of visited states, so it might re-explore the same state via a different path. This can lead to exponential work for problems with loops or redundant paths.
- **Graph Search:** Remembers states that have been visited in an **explored set** (or "closed set"). It avoids expanding states that are already in the explored set, thus handling loops and redundant paths efficiently.

3.2.2 States vs. Nodes

State

Representation of a physical configuration. Describes a specific situation in our environment (e.g., "in Arad").

Node

A data structure to represent a part of a search tree. It includes a **state**, a **parent node**, the **action** taken, the **path cost** ($g(n)$), and the **depth** (e.g., "the path Arad $\rightarrow$ Sibiu").

3.2.3 Key Search Tree Terminology

Fringe

The set of all nodes at the end of all visited paths is called the fringe. (Also known as **frontier** or "open set"). These are the nodes available for expansion.

Depth

Number of levels in the search tree.

3.2.4 Evaluating Search Strategies

Search strategies are evaluated along the following dimensions:

- **Completeness:** Does it always find a solution if one exists?
- **Time Complexity:** Number of node expansions.
- **Space Complexity:** Maximum number of nodes in memory.
- **Optimality:** Does it always find the optimal (least-cost) solution?

Complexity is measured in terms of:

- b: maximum **branching factor** of the search tree.
- d: the **depth** of the optimal solution.
- m: the **maximum depth** of the state space (may be ∞).

3.3 Uninformed Search Strategies

Uninformed Search

Do not have any information about the problem except the problem definition. (Also called **Blind Search**).

Breadth-First Search (BFS)

A special case of Uniform-Cost Search where all step costs are equal. It starts at the tree root and explores the tree **level by level**. It uses a FIFO (First-In-First-Out) queue for the fringe.

- **Completeness:** Yes.
- **Time:** $O(b^d)$ (The summary table uses $O(b^{d+1})$).
- **Space:** $O(b^d)$. Memory consumption is its biggest drawback.
- **Optimality:** Yes (if all costs are equal).

Uniform-Cost Search (UCS)

Each node is associated with a cost, which accumulates over the path. UCS expands the node with the **lowest cumulative path cost** ($g(n)$). It is often implemented with a **priority queue**.

- **Completeness:** Yes (if step costs are positive, i.e., $> \epsilon > 0$).
- **Time:** $O(b^{(}1 + \lfloor C^{/}\epsilon \rfloor))$, where C^* is the cost of the optimal solution.
- **Space:** $O(b^{(}1 + \lfloor C^*/\epsilon \rfloor))$.
- **Optimality:** Yes.

Depth-First Search (DFS)

Starts at the tree root and explores as far as possible along one branch before backtracking. It uses a LIFO (Last-In-First-Out) stack for the fringe.

- **Completeness:** No. Fails in infinite-depth spaces or spaces with loops.
- **Time:** $O(b^m)$, where m is the max depth. Can be terrible if $m \gg d$.
- **Space:** $O(b \times m)$. This linear space complexity is its key advantage.
- **Optimality:** No.

Depth-limited Search (DLS)

A variation of DFS where the search is limited to a predetermined depth l. Nodes at depth l are not expanded.

- **Completeness:** No (if $l < d$).
- **Time:** $O(b^l)$.

- **Space:** $O(b \times l)$.
- **Optimality:** No.

Iterative Deepening Search (IDS)

Combines the benefits of BFS and DFS. It runs DLS repeatedly with increasing depth limits: $l = 0, 1, 2, \ldots, d$.

- **Completeness:** Yes.
- **Time:** $O(b^d)$ (Despite re-generating upper levels, the overhead is small).
- **Space:** $O(b \times d)$.
- **Optimality:** Yes (if costs are uniform).

Bidirectional Search

Performs two searches simultaneously: one forward from the initial state, one backward from the goal state. Stops when the two searches meet.

- **Completeness:** Yes.
- **Time:** $O(b^{d/2})$.
- **Space:** $O(b^{d/2})$.
- **Notes:** Only possible if actions can be reversed.

3.3.1 Summary of Uninformed Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes*	Yes	No	No	Yes*

Table 2: *Comparison of uninformed search strategies. (*Assumes uniform step costs or $l \geq d$ where applicable).*

3.4 Informed Search Strategies

Informed Search

Have additional knowledge about the problem (beyond the definition) and an idea of where to "look" for solutions.

This "hint" is provided by a heuristic function.

3.4.1 Heuristics

Heuristic $h(n)$

Informally denotes a "rule of thumb". In tree-search, a heuristic is a function $h(n)$ that **estimates the remaining cost** to reach the goal from node n.

3.4.2 Greedy Best-first Search

Greedy Best-first Search

Uses an evaluation function $f(n) = h(n)$ to estimate the cost from node n to the goal. It expands the node that appears to be closest to the goal, according to the heuristic. It does not care about the actual cost/distance.

- **Completeness:** No. Can get stuck in loops. (Complete in finite spaces with loop detection).
- **Time:** Worst case $O(b^m)$.
- **Space:** Worst case $O(b^m)$ (keeps all nodes in memory).
- **Optimality:** No. The solution depends entirely on the heuristic.

3.4.3 A* Search

A* Search

An informed tree search algorithm, building on best-first search. It is the "best-known" form. It avoids expanding paths that are already expensive.

A* evaluates nodes using the function: $f(n) = g(n) + h(n)$.

- $g(n) =$ **true cost** so far to reach node n.
- $h(n) =$ **estimated cost** to get from n to the goal.
- $f(n) =$ **estimated cost** of the cheapest solution path that goes through node n.
- **Completeness:** Yes (unless there are infinitely many nodes with $f(n) \leq f(G)$).
- **Time:** Can be exponential unless the error of the heuristic $h(n)$ is bounded.
- **Space:** Has to keep all nodes in memory. This is the primary drawback of A*.
- **Optimality:** Yes, **if the heuristic** $h(n)$ **is admissible**.

3.4.4 Heuristic Properties

Admissible Heuristics

A heuristic is **admissible** if it **never overestimates** the true cost to reach a goal. Formally:

$$h(n) \leq h^*(n) \quad \text{for all nodes } n,$$

where $h(n)$ is the heuristic estimate and $h^*(n)$ is the actual optimal cost from n to the goal. (For example, the straight-line distance heuristic h_{SLD} is admissible in route-finding problems.)

Consistent Heuristics

A heuristic is **consistent** if for every node n and every successor n' generated by action a, the "triangle inequality" holds: $h(n) \leq c(n, a, n') + h(n')$. This means the heuristic difference between adjacent nodes never overestimates the actual step cost.

- **Lemma 1:** Every **consistent** heuristic is also **admissible**.
- **Lemma 2:** If $h(n)$ is consistent, then the values of $f(n)$ along any path are **non-decreasing**.

Relaxed Problems

A problem with fewer restrictions on the actions is called a relaxed problem. The cost of an optimal solution to a relaxed problem is an **admissible heuristic** for the original problem.

Example (8-puzzle):

- $h_1(n)$ = Number of misplaced tiles. (Relaxed rule: tile can move anywhere).
- $h_2(n)$ = Total Manhattan distance. (Relaxed rule: tile can move to any adjacent square).
- Both h_1 and h_2 are admissible.

Dominance

If h_1 and h_2 are both admissible and $h_2(n) \geq h_1(n)$ for all n, then h_2 **dominates** h_1.

A* will expand fewer nodes with a dominant heuristic. (e.g., for the 8-puzzle, h_2 (Manhattan) dominates h_1 (misplaced tiles)).

Combining Heuristics If we have several admissible heuristics $h_1(n), \ldots, h_m(n)$, we can combine them. $h(n) = \max h_1(n), h_2(n), \ldots, h_m(n)$ is also admissible and dominates all of its components.

3.4.5 Optimality of A

A (using Tree Search) is optimal if its heuristic $h(n)$ is admissible.

Proof (Informal):

1. Let G be an optimal goal state, with path cost C^*.
2. Assume for contradiction that A is about to return a suboptimal goal G_2, with path cost $g(G_2) > C^*$.
3. At this moment, G_2 is in the fringe. Because A^* chose G_2, its f-value must be the lowest, so $f(G_2) \leq f(n)$ for all other fringe nodes n.
4. Let n be any unexpanded node on a true optimal path to G. This node n must be in the fringe.
5. **Analyze $f(G_2)$:** For a goal state, $h(G_2) = 0$. So, $f(G_2) = g(G_2) + h(G_2) = g(G_2)$. Since G_2 is suboptimal, $f(G_2) = g(G_2) > C^*$.
6. **Analyze $f(n)$:** $f(n) = g(n) + h(n)$. Because h is admissible, $h(n) \leq h^(n)$ (where $h^(n)$ is the true cost from n to G). The true cost of the optimal path is $C^= g(n) + h^(n)$. Therefore, $f(n) = g(n) + h(n) \leq g(n) + h^(n) = C^*$.
7. **Contradiction:** We have shown $f(n) \leq C^*$ and $f(G_2) > C^*$. This means $f(n) < f(G_2)$. A would be forced to expand n (on the optimal path) *before* it could ever expand G_2. Thus, A* can never select a suboptimal goal. It is optimal.

3.4.6 Memory-Bounded Heuristic Search

The main problem with A* is its space complexity. Alternatives include:

1. **Iterative-deepening A* (IDA):** Like IDS, but the "depth" cutoff is the f-cost $(g + h)$.
2. **Recursive best-first search (RBFS):** Mimics best-first search using linear space by recursively re-expanding nodes and updating f-values from ancestors.
3. **(Simple) Memory-bounded A ((S)MA*):** When memory is full, drops the worst (highest f-value) leaf node.

3.5 Tree Search vs. Graph Search (Revisited)

Failure to detect repeated states can turn a linear problem into an exponential one!

Graph Search

Uses an **explored set** (or "closed set") to store all states that have been expanded. When expanding a node, its successors are only added to the fringe **if they are not in the fringe or explored set**.

Optimality of A* Graph Search

- If $h(n)$ is only **admissible**, Graph Search A* is not guaranteed to be optimal. It might find a suboptimal path to a node first, add it to the explored set, and never find the optimal path.
- If $h(n)$ is **consistent**, Graph Search A* **is optimal**.
- **Why?** A consistent heuristic guarantees that f-values are non-decreasing along any path. This means the *first* time A* expands a node n, it is *guaranteed* to have found the shortest possible path to it. Therefore, we never need to re-expand any node in the explored set.

4 Local Search and Adversarial Search

In previous lectures, the focus was on finding a sequence of actions (a path) to reach a goal (e.g., A* search). However, in many AI problems, the path to the solution is irrelevant; only the final state matters.

Optimization Problem

An **Optimization Problem** is a problem where all states are technically solutions, but they have different qualities. The goal is not to reach a state, but to find the **best state** (Global Optimum) according to an **Objective Function**.

The State Landscape Imagine the state space as a physical landscape where "elevation" corresponds to the value of the objective function.

- **Global Maximum**: The highest peak in the entire state space. The optimal solution.
- **Local Maximum**: A peak that is higher than all its immediate neighbors but lower than the global maximum.
- **Shoulder**: A flat region (plateau) that can be an intermediate step rising to a peak.
- **Flat Local Maximum**: A plateau that is a local maximum.

4.1 Local Search Algorithms

Local search algorithms operate by maintaining a single **Current State** and iteratively moving to neighboring states.

- **Advantages:** Uses very little memory (constant space) and can find reasonable solutions in infinite/massive search spaces.
- **Disadvantages:** No guarantee of completeness (finding a solution) or optimality (finding the best solution).

4.1.1 Hill Climbing (Greedy Local Search)

The Hill Climbing algorithm is a loop that continually moves in the direction of increasing value (uphill).

1. Generate all neighbors of the current state.
2. Move to the neighbor with the highest value.
3. Terminate when no neighbor has a higher value than the current state.

Metaphor

Hill climbing is like "climbing Mount Everest in thick fog with amnesia." You only know the slope under your feet, not where the true peak is.

Problems with Hill Climbing

- **Local Optima**: The algorithm halts at a local peak, missing the higher global peak elsewhere.
- **Ridges**: A sequence of local maxima that is difficult for greedy algorithms to navigate. If the ridge rises diagonally but movement is restricted to cardinal directions (North/East), the algorithm may think it's at a peak because every single step leads downhill, even though the ridge continues up.
- **Plateaus**: Flat areas where the algorithm cannot determine which direction is better, leading to random wandering.

Variants to Solve Local Optima

- **Randomized Restart**: Run the algorithm multiple times from different random starting positions.
- **Stochastic Hill Climbing**: Choose a successor randomly, weighted by the steepness of the uphill move (better moves have higher probability).

4.1.2 Beam Search

Instead of keeping just one state in memory, **Beam Search** keeps track of k states.

- At each step, generate all successors of all k states.
- Select the best k successors from the complete list to be the new current states.
- This is different from k random restarts because the "beams" (threads) can communicate/converge on the most promising area.

4.1.3 Simulated Annealing

This algorithm combines Hill Climbing with a Random Walk to escape local optima. It is inspired by metallurgy (heating metal and cooling it slowly to toughen it).

Mechanism

Instead of always picking the best move, the algorithm sometimes accepts "bad" moves (downhill) to explore the space. The probability of accepting a bad move depends on the **Temperature (T)**.

The Process:

- **High T (Early phase):** The algorithm accepts bad moves frequently (exploring widely, behaving like a random walk).
- **Low T (Late phase):** The algorithm rarely accepts bad moves (exploiting the local area, behaving like hill climbing).

The Probability Formula: If the move improves the situation ($\Delta E > 0$), accept it always. If the move is worse ($\Delta E < 0$), accept it with probability:

$$P = e^{\frac{\Delta E}{T}}$$

As $T \to 0$, the probability of accepting a negative ΔE approaches 0.

4.2 Gradient Descent

When the state space is **continuous** (rather than discrete steps), we use Gradient Descent. This is the foundation of training Neural Networks.

Gradient

The gradient $\nabla J(\theta)$ is a vector of partial derivatives pointing in the direction of the steepest ascent.

To minimize a cost function $J(\theta)$:

$$\theta_{new} = \theta_{old} - \lambda \nabla J(\theta)$$

Where λ is the **Learning Rate**.

4.2.1 The Learning Rate (λ)

This hyperparameter controls the step size.

- **Too Small:** Convergence is extremely slow; might get stuck in local minima.
- **Too Large:** The algorithm may overshoot the minimum, oscillate, or diverge (move away from the solution).

4.3 Brief Note: SAT and Complexity

The Boolean Satisfiability Problem (SAT) is finding an assignment of variables that makes a logical formula true.

- **Cook's Theorem**: SAT is NP-Complete.
- **NP-Complete**: Problems that are effectively "hardest" in NP. If you can solve one NP-Complete problem in polynomial time, you can solve all of them ($P = NP$).
- **GSAT:** A local search algorithm for SAT that flips variable assignments to minimize unsatisfied clauses.

4.4 Adversarial Search (Games)

This domain deals with multi-agent environments where agents have conflicting goals (Zero-Sum Games).

Zero-Sum Game

A situation where one player's gain is exactly the other player's loss. The total utility remains constant (usually 0).

4.4.1 Game Trees

Unlike standard search trees, nodes represent game states, and levels alternate between players:

- **MAX**: The agent trying to maximize the utility value (Us).
- **MIN**: The opponent trying to minimize the utility value (Enemy).

4.4.2 The Minimax Algorithm

The Minimax algorithm determines the optimal move by performing a Depth-First Search (DFS) to the end of the game tree and then reasoning **backwards** (bottom-up).

[Image of minimax game tree example]

The Process:

1. **Dive (DFS):** Traverse the tree all the way down to the leaf nodes (terminal states).
2. **Evaluate:** Assign utility values to the leaf nodes (e.g., Win=1, Loss=-1, or numerical scores).
3. **Propagate (Backtracking):**
   - If the parent is a **MAX** node: It looks at its children and picks the **highest** value.
   - If the parent is a **MIN** node: It looks at its children and picks the **lowest** value.
4. **Root Decision:** When the values reach the root, MAX chooses the move leading to the highest value.

Note: If multiple moves have the same value, standard implementations often pick the one found first (left-most).

Complexity:

- Time: $O(b^m)$ (Exponential). b is branching factor, m is max depth.
- Space: $O(bm)$ (Linear), due to Depth First Search nature.

4.5 Alpha-Beta Pruning

Because Game Trees grow exponentially (10^{40} nodes for Chess), we cannot search the whole tree. **Pruning** allows us to ignore branches that will certainly not be chosen, without affecting the final result.

4.5.1 Concept: Alpha and Beta

We maintain two values during the search that represent the "best guarantees" found so far on the path from the root.

- α **(Alpha)**: The best (highest) value **MAX** can guarantee so far.
  - Initialized to $-\infty$.
  - Updated only at **MAX** nodes (when a child returns a value higher than current α).
- β **(Beta)**: The best (lowest) value **MIN** can guarantee so far.
  - Initialized to $+\infty$.
  - Updated only at **MIN** nodes (when a child returns a value lower than current β).

4.5.2 The Pruning Condition

A branch is pruned (cut off) immediately when:

$$\alpha \geq \beta$$

Why does this work?

- **Beta Cutoff (at MIN node):** If MIN finds a move that results in a value v where $v \leq \alpha$, MIN will choose that (or something even lower). However, the MAX player at the parent node already has a guaranteed option worth α from a previous branch. MAX will therefore **never** choose the path to this MIN node. The rest of MIN's children are irrelevant.
- **Alpha Cutoff (at MAX node):** Symmetrically, if MAX finds a move worth $v \geq \beta$, MIN (at the parent node) will never allow the game to reach this MAX node, because MIN already has a better option (worth β) elsewhere.

Efficiency: With optimal move ordering (checking best moves first), Alpha-Beta pruning effectively doubles the searchable depth, reducing complexity to $O(b^{m/2})$.

4.6 Advanced Game Search

4.6.1 Evaluation Functions

Since we cannot reach terminal nodes in complex games (Chess/Go), we cut off the search at a specific depth and use a **Heuristic Evaluation Function**. This estimates the probability of winning from that state (e.g., Material value in chess: Pawn=1, Queen=9).

4.6.2 Monte Carlo Tree Search (MCTS)

Used in games like Go where branching factors are too high for Minimax ($b \approx 250$).

- Instead of exhaustive search, it plays many random "simulation" games from the current state to the end.
- It builds statistics: "In 80% of random games starting from move A, I won."
- **AlphaGo:** Combined MCTS with Deep Neural Networks (to predict move probabilities and value positions) to defeat the world champion.

5 Constraint Satisfaction Problems

In standard search problems (like pathfinding), we care about the sequence of actions (the path) to reach a goal. In **Constraint Satisfaction Problems (CSPs)**, the path is irrelevant. We only care about the **goal state** itself.

Definition: CSP

A CSP is defined by three components:

- **Variables (X):** A set of variables $\{X_1, \ldots, X_n\}$.
- **Domains (D):** A set of domains $\{D_1, \ldots, D_n\}$, where each variable X_i has a set of possible values D_i.
- **Constraints (C):** A set of constraints specifying allowable combinations of values for subsets of variables.

5.1 Types of Assignments

- **Partial Assignment:** Values are assigned to only some variables.
- **Consistent (Legal) Assignment:** An assignment that does not violate any constraints.
- **Complete Assignment:** Every variable is assigned a value.
- **Solution:** A consistent, complete assignment.

5.2 Constraint Graphs

Problems are often visualized as a **Constraint Graph**:

- **Nodes:** Represent variables.
- **Edges:** Represent constraints between variables.

This abstraction helps us understand the structure of the problem (e.g., independent subproblems).

5.3 Types of Constraints

1. **Unary Constraint:** Restricts the value of a single variable (e.g., $SA \neq$ green).
2. **Binary Constraint:** Relates two variables (e.g., $SA \neq WA$).
3. **Higher-order Constraint:** Involves 3 or more variables (e.g., Cryptarithmetic puzzles like $TWO + TWO = FOUR$).
4. **Soft Constraints (Preferences):** Constraints that are not binding but preferred (e.g., "Red is better than Green"). These turn the CSP into a **Constrained Optimization Problem**.

5.4 Solving CSPs: Search Strategies

We can view CSPs as a search problem where the initial state is empty, and the successor function assigns a value to an unassigned variable.

5.4.1 Naïve Search vs. Commutativity

A naïve Breadth-First or Depth-First search would branch on every variable and every value, leading to a massive search space ($n! \cdot v^n$). However, CSP assignments are **commutative**. The order in which we assign variables does not matter ($WA =$ red then $NT =$ green is the same state as $NT =$ green then $WA =$ red).

- **Implication:** We only need to consider assigning a value to *one* variable at each node.
- This reduces the search space to v^n (where v is domain size, n is number of variables).

5.4.2 Backtracking Search

Backtracking is the fundamental uninformed search algorithm for CSPs. It performs a Depth-First Search (DFS).

Backtracking Logic

1. Select an unassigned variable.
2. Try a value from its domain.
3. If the value is consistent with current assignments, proceed recursively.
4. If a conflict arises, **backtrack** (undo the assignment and try the next value).
5. If all values fail, return failure.

5.5 Heuristics: Improving Backtracking

To make backtracking efficient, we need to make smart decisions about *which* variable to assign next and *which* value to try first.

5.5.1 Variable Selection Heuristics (Which variable next?)

1. **Minimum Remaining Values (MRV):**
   - Also known as the "Fail-First" heuristic.
   - **Rule:** Choose the variable with the *fewest* legal values remaining in its domain.
   - **Reasoning:** If a variable has only 1 option left, we should assign it immediately to detect inevitable failures early.
2. **Degree Heuristic:**
   - Used as a tie-breaker for MRV.
   - **Rule:** Choose the variable involved in the largest number of constraints on *unassigned* variables.
   - **Reasoning:** Assigning this variable reduces the branching factor of future steps the most.

5.5.2 Value Selection Heuristics (Which value first?)

1. **Least Constraining Value (LCV):**
   - **Rule:** Given a variable, choose the value that rules out the *fewest* choices for the neighboring variables.
   - **Reasoning:** We want to leave maximum flexibility for subsequent assignments to find a solution.

5.6 Constraint Propagation

Search implies "trying" values. Constraint propagation implies "inferring" which values are impossible and removing them before search to reduce the search space.

5.6.1 Levels of Consistency

- **Node Consistency:** Every single variable satisfies its unary constraints.
- **Arc Consistency (2-Consistency):** A variable X is arc-consistent with respect to Y if for every value $x \in D_X$, there exists some value $y \in D_Y$ that satisfies the binary constraint.
- **Path Consistency (3-Consistency):** Looks at triples of variables.
- **k-Consistency:** Generalization to k variables. Checking high k is computationally expensive ($O(d^k)$).

5.6.2 Algorithms for Propagation

Forward Checking When variable X is assigned a value:

1. Look at all unassigned neighbors Y.
2. Remove any values from D_Y that conflict with the assignment of X.
3. If any domain becomes empty, backtrack immediately.

Limitation: It only checks immediate consequences of an assignment. It fails to detect failures further down the chain (e.g., if two neighbors are forced to take the same value, Forward Checking won't see it until one is assigned).

Arc Consistency (AC-3 Algorithm) This is more powerful than Forward Checking. It propagates constraints through the whole graph until stability.

AC-3 Algorithm

1. Initialize a **Queue** with all arcs (constraints) in the CSP.
2. While the Queue is not empty:
   - Remove an arc (X_i, X_j).
   - Check if X_i is consistent with X_j.
   - If we remove a value from D_i to make it consistent:
     - We must re-check all neighbors of X_i (add arcs (X_k, X_i) back to the Queue).

5.7 Local Search for CSPs

Instead of building a partial assignment (constructive search), Local Search starts with a complete (but likely inconsistent) assignment and tries to repair it.

- **State:** Complete assignment of all variables.
- **Goal:** Eliminate all violated constraints.
- **Min-Conflicts Heuristic:**
  1. Choose a random variable that is currently involved in a conflict.
  2. Change its value to the one that violates the *fewest* constraints (minimizes conflicts).

This is surprisingly effective for problems like N-Queens (can solve for millions of queens).

5.8 Problem Structure and Decomposition

5.8.1 Independent Subproblems

If the constraint graph has connected components that are not connected to each other, we can solve them separately. This reduces complexity from $O(d^n)$ to $O(n/c \cdot d^c)$ where c is the size of the subproblem.

5.8.2 Tree-Structured CSPs

If the constraint graph is a tree (no loops), the CSP can be solved in **linear time** $O(n \cdot d^2)$.

Algorithm for Tree CSPs:

1. **Topological Sort:** Pick a root and order variables such that parents appear before children $(X_1 \dots X_n)$.
2. **Backward Pass (Constraint Propagation):** From X_n down to X_2, make each parent arc-consistent with its child. This removes all inconsistent values.
3. **Forward Pass (Assignment):** From X_1 to X_n, assign any valid value consistent with the parent. Since the graph is arc-consistent, a solution is guaranteed without backtracking.

5.8.3 Nearly Tree-Structured Problems

Most real problems are not trees, but we can transform them.

Cutset Conditioning Idea: Remove a set of variables (the **Cycle Cutset**) so that the remaining graph is a tree.

1. Identify a subset of variables S (the cutset).
2. Assign values to variables in S.
3. These assignments act as constraints on the remaining variables.
4. Solve the remaining (now tree-structured) problem efficiently.
5. If no solution, try a different assignment for S.

6 Introduction: Logic and AI

Artificial Intelligence aims to create agents that not only search for solutions but "understand" the world. While search algorithms generate successors and evaluate states, they lack a representation of knowledge. **Logic** provides the framework for this representation.

6.1 Knowledge-Based Agents

A **Knowledge-Based Agent** maintains a representation of the world and uses logical reasoning to derive new information and make decisions. It operates on two main components:

Components of a Knowledge-Based System

- **Knowledge Base (KB)**: A set of sentences in a formal language representing facts about the world. It follows a declarative approach (TELL the agent what it needs to know).
- **Inference Engine**: Domain-independent algorithms that derive new sentences (conclusions) from the KB.

The interaction loop of a KB-Agent involves:

1. **TELL**: The agent incorporates new percepts into the KB.
2. **ASK**: The agent queries the KB to decide on an action.
3. **TELL**: The agent records the chosen action and updates the time.

6.2 The Wumpus World

The Wumpus World is a standard environment used to illustrate logical reasoning in AI. It is a grid-based cave where an agent must find gold while avoiding pits and a monster (Wumpus).

6.2.1 PEAS Description

- **Performance**: $+1000$ for gold, -1000 for death, -1 per step, -10 for using the arrow.
- **Environment**:
  - Squares adjacent to the **Wumpus** smell (Stench).
  - Squares adjacent to a **Pit** are breezy (Breeze).
  - Gold glitters in its square.
- **Sensors**: $[Stench, Breeze, Glitter, Bump, Scream]$.
- **Actuators**: Turn Left/Right, Forward, Grab, Release, Shoot.

6.2.2 Reasoning Example

If the agent is in $[1,1]$ and perceives no breeze and no stench, it knows $[1,2]$ and $[2,1]$ are safe (OK). If it moves to $[2,1]$ and perceives a breeze, it infers a pit must be in $[2,2]$ or $[3,1]$. Logic allows the agent to combine observations over time to build a map of safe and dangerous areas.

6.3 Propositional Logic (PL)

Propositional logic is the simplest logic, where symbols represent whole propositions (facts) that can be true or false.

6.3.1 Syntax

Syntax defines the rules for constructing well-formed sentences. We use **Backus-Naur Form (BNF)**:

- **Atomic Sentences**: Single symbols (e.g., P, Q, $RoommateWet$).
- **Complex Sentences**: Constructed using logical connectives.
  - $\neg P$ (Not/Negation)
  - $P \wedge Q$ (And/Conjunction)
  - $P \vee Q$ (Or/Disjunction)
  - $P \Rightarrow Q$ (Implication/If-Then)
  - $P \Leftrightarrow Q$ (Biconditional/If and only if)

6.3.2 Semantics

Semantics defines the meaning of sentences, specifically their **Truth Value** relative to a specific world configuration (Interpretation).

Model

A **Model** is an interpretation (a specific setting of true/false values for all propositional symbols) in which a specific sentence or Knowledge Base is **True**.

Truth Tables The semantics are defined by truth tables. Key logical behaviors to remember:

- $P \wedge Q$ is true only if *both* are true.
- $P \vee Q$ is true if *at least one* is true (inclusive OR).
- $P \Rightarrow Q$ is true unless P is true and Q is false. (Note: $False \Rightarrow True$ is valid/True).

6.3.3 Logical Properties

1. **Tautology**: A sentence that is true in *all* possible models (e.g., $P \vee \neg P$).
2. **Satisfiability**: A sentence is satisfiable if it is true in *at least one* model.
3. **Contradiction**: A sentence that is false in all models (e.g., $P \wedge \neg P$).
4. **Logical Equivalence**: Two sentences α and β are equivalent ($\alpha \equiv \beta$) if they have the same truth value in every model.

Important Equivalences (for simplification)

- **Double Negation**: $\neg(\neg A) \equiv A$
- **Contraposition**: $(A \Rightarrow B) \equiv (\neg B \Rightarrow \neg A)$
- **Implication Elimination**: $(A \Rightarrow B) \equiv (\neg A \vee B)$
- **De Morgan's Laws**:
  - $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
  - $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$
- **Distributivity**: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

6.4 Inference and Entailment

The core goal of the Inference Engine is to determine if a sentence follows from the Knowledge Base.

Entailment ($KB \models \alpha$)

We say the Knowledge Base KB **entails** sentence α if and only if α is true in all models where KB is true.

6.4.1 Model Checking

A simple algorithm to check entailment is **Truth Table Enumeration**:

1. Enumerate all possible assignments of True/False to the symbols.
2. Check if the KB is true in that assignment.
3. If KB is true, check if α is also true.
4. If α is true in every model where KB is true, then $KB \models \alpha$.

Drawback: The time complexity is $O(2^n)$, making it inefficient for large numbers of variables.

6.4.2 Consistency and The Principle of Explosion

It is critical that a Knowledge Base is **Consistent**.

- If a KB contains a contradiction (e.g., P and $\neg P$), it is inconsistent.
- An inconsistent KB entails **everything**.
- *Example*: If "The roommate flies" and "The roommate does not fly" are both in the KB, we can prove "The Moon is made of cheese."
- This relies on the **Law of Non-Contradiction** (Aristotle): A and $\neg A$ cannot both be true.

6.5 Resolution and Proof Systems

To avoid enumerating truth tables, we use syntactic proof systems like **Resolution**. To use resolution, sentences must be in a specific form.

6.5.1 Conjunctive Normal Form (CNF)

Any sentence in propositional logic can be converted into CNF. A CNF formula is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals.

$$(L_{1,1} \vee \cdots \vee L_{1,k}) \wedge (L_{2,1} \vee \dots) \wedge \dots$$

where a literal is a symbol (P) or its negation ($\neg P$).

Conversion Steps (Example):

1. Eliminate $\Leftrightarrow$ and $\Rightarrow$ using $(A \Rightarrow B) \equiv (\neg A \vee B)$.
2. Move $\neg$ inwards using De Morgan's Laws.
3. Distribute $\vee$ over $\wedge$.

6.5.2 The Resolution Rule

The resolution inference rule takes two clauses and produces a new one:

$$(P \vee A) \text{ and } (\neg P \vee B) \text{ derive } (A \vee B)$$

Here, P and $\neg P$ are complementary literals. They "cancel out."

Unit Resolution

A simplified version where one clause is a single literal:

$$(l_1 \vee \cdots \vee l_k) \text{ and } \neg l_i \implies (l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \dots)$$

This is effectively **Modus Ponens** in CNF form.

6.5.3 Proof by Refutation (Resolution Algorithm)

To prove that $KB \models \alpha$, we use **Proof by Contradiction**:

1. Assume $\neg\alpha$ (the negation of what we want to prove).
2. Add $\neg\alpha$ to the Knowledge Base.
3. Convert the entire set of sentences to CNF.
4. Repeatedly apply the Resolution Rule to pairs of clauses containing complementary literals.
5. If you derive the **Empty Clause** (a contradiction, e.g., resolving P and $\neg P$), then the original assumption $\neg\alpha$ must be false, meaning α is true.

6.6 Horn Clauses

Resolution is complete (can prove anything that is true) but can be slow (exponential in worst case). **Horn Clauses** are a subset of PL that allows for more efficient inference.

- A Horn Clause is a disjunction of literals with **at most one positive literal**.
- *Example*: $\neg P \vee \neg Q \vee R$ is equivalent to $(P \wedge Q) \Rightarrow R$.
- Inference with Horn Clauses can be done in linear time using **Forward Chaining**.

6.7 Limitations of Propositional Logic

While powerful, PL has distinct limitations:

1. **Lack of Objects and Relations**: In PL, "Roommate carrying umbrella" is a single atomic symbol. The logic does not understand that "Roommate" is a person or "Umbrella" is an object.
2. **Verbose**: To say "All pits cause breezes," we must write a rule for every square: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$, $B_{1,2} \Leftrightarrow \dots$
3. **Variable Explosion**: In the Wumpus world alone, a small grid generates 64 distinct symbols and hundreds of sentences.

These limitations lead to the development of **First-Order Logic** (not covered in this summary).