# Foundations of AI - SS25

Niklas Rodenbüsch

August 14, 2025

## 1 Rational Agents

- **Agent**: Perceives environment through sensors and acts upon it through actuators. Consists of an *agent program* running on an *architecture*.
- **Ideal rational agent**: does the "right thing" in terms of a *performance measure*.
- **Rationality** is distinct from **omniscience**. An omniscient agent knows the actual effects of its actions, whereas a rational agent acts to maximize its *expected* performance based on its percepts and knowledge .

**Definition 1:**

For each possible percept sequence, a **rational agent** should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

### 1.1 Classes of Agents

- **Table-Driven:** Use a lookup table indexed by the entire percept sequence to select an action.
- **Simple Reflex:** React based only on the current percept, using a set of condition-action rules.
- **Model-based Reflex:** Maintain an internal *state* about the world and the effects of its own actions.
- **Goal-based:** Use explicit *goals* to decide on actions, considering how a potential action will bring them closer to their goal.
- **Utility-based:** When there are multiple possible next actions, a *utility function* is used to map a state to a real number.
- **Learning:** Can improve their performance over time by modifying their behavior. They consist of four main components:
  - **Performance element:** Selects external actions.
  - **Critic:** Provides feedback on how well the agent is doing based on a performance standard.
  - **Learning element:** Makes improvements to the agent's performance capabilities.
  - **Problem generator:** Suggests actions that can lead to new and informative experiences.

### 1.2 Types of Environments

The nature of the environment determines the required complexity of the agent. Some environments are more demanding than others.

**Most challenging**: Partially observable, nondeterministic, strategic, dynamic, continuous and multi-agent.

# 2 Solving Problems by Searching

Problem-solving agents are a type of goal-based agent that formulates a problem in terms of a *state-space* and a *goal-state*. Given an *initial state* the agent's objective is to find a sequence of actions that leads to the desired goal state. This approach typically assumes the environment is fully observable, deterministic, static, discrete, and single-agent.

## 2.1 Concepts

- **Transition Model:** Description of the outcome of an action (successor function).
- **Goal Test:** Tests whether the state description matches a goal state.
- **Path:** A sequence of actions leading from one state to another.
- **Problem Type:** Depends on the knowledge of the world states and actions.
- **Costs:** Specification of search costs (search costs, offline costs) and execution costs (path costs, online costs).
- **Search Cost:** Time and storage requirements to find a solution.
- **State Space:** The set of all possible states the environment can be in. This is an abstraction of the real world, containing only relevant details.
- **Actions:** A set of possible actions available to the agent that can change the world state. Availability of actions might be a function of the state.

## 2.2 Problem Formulation

The way a problem is formulated can significantly influence the difficulty of finding a solution! A problem is formally defined as a 5-tuple of: **state space, initial state, actions, goal test** and **path costs**.

## 2.3 Problem Types

The nature of a problem depends on the agent's knowledge of states and actions.

- **Completely observable state:** Agent has complete world state and action knowledge. Solution is reduced to searching for a path from the initial state to a goal state.
- **Partially observable state:** Incomplete world state and action knowledge. The agent only knows which group of world states it is in.
- **Contingency Problem:** Arises when the choice of action depends on information that will only be available at execution time. The solution must include branching based on (possible) future percepts.
- **Exploration Problem:** State space and the effects of actions are unknown to the agent.

## 2.4 Search Strategies

Search algorithms are evaluated based on four criteria:

- **Completeness:** Is the algorithm guaranteed to find a solution if one exists?
- **Optimality:** Does the algorithm find the solution with the lowest path cost?
- **Time & Space Complexity**

These are often measured in terms of $b$ (branching factor), $d$ (depth of the shallowest goal), and $m$ (maximum path length in the state space).

## 2.5 Uninformed (Blind) Search

Uninformed search strategies use no information about the distance or cost to the goal.

- **Breadth-First Search (BFS):** Expands nodes in the order they were generated, exploring layer by layer (FIFO queue). It is complete and is optimal if all action costs are identical. Both time and space complexity are $O(b^d)$.
- **Uniform-Cost Search (UCS):** Expands the node with the lowest path cost, $g(n)$, using a priority queue. It finds the optimal solution provided action costs are non-negative.
- **Depth-First Search (DFS):** Always expands the deepest node first (LIFO queue). It is neither complete nor optimal in general. Its primary advantage is its modest space complexity of $O(bm)$ in tree-based searches, as it only needs to store the current path.
- **Depth-Limited Search (DLS):** A variation of DFS that imposes a cutoff on the maximum search depth to prevent infinite paths.
- **Iterative Deepening Search (IDS):** Combines the benefits of BFS and DFS by running DLS with progressively increasing depth limits. It is complete and optimal (like BFS) but with the low space complexity of DFS ($O(bd)$). It is often the preferred uninformed search method when the search space is large and the solution depth is unknown.
- **Bidirectional Search:** Simultaneously searches forward from the initial state and backward from the goal state. This can drastically reduce time complexity to $O(b^{d/2})$. However, it requires reversible operators and an easily definable set of goal states.

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes[a] | Yes[a,b] | No | No | Yes[a] | Yes[a,d] |
| Time | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(bl)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes[c] | Yes | No | No | Yes[c] | Yes[c,d] |

$b$ branching factor
$d$ depth of solution
$m$ maximum depth of the search tree
$l$ depth limit
$C^*$ cost of the optimal solution
$\epsilon$ minimal cost of an action

Superscripts:
[a] $b$ is finite
[b] if step costs not less than $\epsilon$
[c] if step costs are all identical
[d] if both directions use breadth-first search

# 3 Constraint Satisfaction Problems (CSPs)

A Constraint Satisfaction Problem (CSP) is a specialized type of search problem where states are defined by assignments of values to a set of variables, and the goal is to satisfy a set of constraints on these variables.

**Definition 2:**

A **Constraint Satisfaction Problem** is defined by three components :

- A set of variables $\{x_1, x_2, ..., x_n\}$.
- A set of domains (values) for each variable $\{dom_1, dom_2, ..., dom_n\} \rightarrow d^n$ possible assignments
- A set of constraints that specify allowable combinations of values for subsets of variables.

A solution to a CSP is a complete assignment of values to all variables such that all constraints are satisfied. For binary constraints, the problem can be visualized as a **constraint graph**, where nodes are variables and edges represent constraints between them.

## 3.1 Backtracking Search

CSPs are often solved using **backtracking search**, which is a form of depth-first search. The algorithm incrementally assigns values to variables one at a time, and "backtracks" when a variable has no legal values left to assign.

## 3.2 Heuristics & Pruning Techniques

Goal: Try to keep the search tree as small as possible and to exploit the CSP problem structure.

- **Variable Ordering Heuristics:**
  - **Most-Constrain*ed*-Variable:** Choose the variable with the fewest remaining legal values. Helps to detect failures early and directly reduces branching factor.
  - **Most-Constrain*ing*-Variable:** As a tie-breaker among variables with the same number of remaining legal values, select the variable that is involved in the most constraints on other unassigned variables. This helps to prune the search tree in future steps.
- **Value Ordering Heuristics:**
  - **Least-Constraining-Value:** For a given variable, prefer the value that rules out the fewest choices for the neighboring variables. This leaves more options available, increasing the chance of finding a solution on the current path.
- **Inference and Constraint Propagation:**
  - **Forward Checking:** After assigning a value to a variable, check its neighbors and delete any values from their domains that are now inconsistent. If any variable's domain becomes empty, the search can backtrack immediately.
  - **Arc Consistency:** A stronger form of inference. An arc from variable $X$ to $Y$ is consistent *iff*, for every $x \in X$, there is a corresponding value $y \in Y$ that satisfies the constraint. The **AC-3 algorithm** ($O(d^3n^2)$) enforces arc consistency throughout the graph. It can be used as a preprocessing step or during search to prune the search space.
    **Note:** Detecting all inconsistencies is NP-hard!

### 3.3 Exploiting Problem Structure

The structure of the constraint graph can be used to find solutions more efficiently.

- **Disconnected Components:** If the constraint graph has multiple disconnected components, each can be solved as an independent subproblem.
- **Tree-Structured CSPs:** If the constraint graph is a tree (i.e., has no loops), the CSP can be solved in polynomial time ($O(nd^2)$). This is achieved by ordering the nodes topologically, enforcing arc consistency, and then assigning values from the root down without any need for backtracking.
- **Almost Tree-Structured CSPs:** For problems that are nearly trees, one can use:
  - **Cutset Conditioning:** A small set of variables (a "cycle cutset") is chosen to break all loops in the graph. The algorithm tries all consistent assignments for the cutset variables; for each assignment, the remaining problem is a tree and can be solved efficiently.
  - **Tree Decomposition:** The problem is decomposed into a set of connected subproblems, which are organized into a tree. The subproblems are solved, and the solutions are then combined. The efficiency of this method depends on the problem's *tree width* $w$.

## 4 Informed Search Methods

Informed (or heuristic) search strategies use problem-specific knowledge to find solutions more efficiently than uninformed methods. This knowledge is supplied by a **heuristic function** $h(n)$, which estimates the cost of the cheapest path from a node $n$ to a goal.

### 4.1 Best-First Search

Best-first search is a general search algorithm that selects the next node to expand based on an **evaluation function** $f(n)$. It uses a priority queue to always expand the node with the "best" (lowest) $f$-value. Different informed search strategies are instances of best-first search with different evaluation functions.

#### 4.1.1 Greedy Best-First Search

Greedy search uses only the heuristic function for evaluation: $f(n) = h(n)$. It tries to expand the node that it estimates to be closest to the goal. While this approach is often fast, it is not optimal and is generally incomplete.

#### 4.1.2 A* Search

A* search combines the benefits of Uniform-Cost Search and Greedy Search. Its evaluation function balances the cost to reach the current node with the estimated cost to get to the goal:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the known cost of the path from the initial state to node $n$, and $h(n)$ is the heuristic estimate of the cost from $n$ to the goal.
A* is guaranteed to be **complete and optimal** if its heuristic function $h(n)$ is **admissible**, meaning it never overestimates the true cost to reach the goal ($h(n) \leq h^*(n)$). A stronger condition, **consistency**, is required for the graph-search version of A* to be optimal without re-opening already visited nodes (consistency implies admissibility). Graph-A* can still be applied if $h(n)$ is not consistent, but optimality is lost. Drawback: Exponential space complexity, as it must store all generated nodes.

**Iterative Deepening A* (IDA*)** is a variant that uses the f-cost as a cutoff instead of depth, combining the optimality of A* with the low memory requirements of iterative deepening.

### 4.2 Local Search Methods

For problems where the path to the solution is irrelevant (e.g., 8-queens), local search algorithms can be very effective. These methods operate on a single current state, moving to neighboring states to optimize an objective function. They require very little memory.

- **Hill-Climbing:** A simple local search that always moves to the best neighboring state. It can get stuck in **local maxima**, on **plateaus** (flat areas where no neighbor is better), and on **ridges**. Can be mitigated by starting over or by injecting noise (random walk).
- **Simulated Annealing:** An improvement on hill-climbing that can escape local maxima. It allows for "bad" (downhill) moves with a probability (noise) that decreases over time. This probability is controlled by a "temperature" parameter $T$, which is gradually lowered according to a schedule.

### 4.3 Genetic Algorithms

Genetic Algorithms (GAs) are a class of local search methods inspired by natural evolution. They maintain a **population** of candidate solutions (individuals), which are evolved over generations. The core components are:

- **Fitness Function:** An objective function that evaluates the quality of each individual in the population.
- **Selection:** Fitter individuals are more likely to be selected to produce offspring.
- **Crossover:** Creates new individuals by combining genetic material (e.g., parts of bit-strings) from two parents.
- **Mutation:** Randomly alters parts of an individual, introducing new traits and preventing premature convergence.

## 5 Board Games

Board games represent a classic area of AI research, framing the competition between two opponents as a search problem. While the states in a game are fully accessible, they are also contingency problems because a player cannot control the opponent's moves. The primary challenge is the massive size of the game's state space.

**Definition 3:**

A game can be defined as a 4-tuple of **initial state**, **operators** (legal moves), a **terminal test** and a **utility function** (outcome of the game).

### 5.1 Minimax Search

For two-player, zero-sum games (with players typically called **MAX** and **MIN**), the **minimax algorithm** can determine the optimal move. It assumes that both players play optimally. The algorithm explores the entire game tree to find a move that maximizes the utility for MAX, while assuming MIN will always try to minimize it:

The algorithm generates the entire game tree using DFS. A **utility function** assigns a value to terminal states of the game (win, loss, or draw). The algorithm propagates these values up the tree: MAX nodes take the maximum value of their children, and MIN nodes take the minimum. MAX then selects the move at the root that leads to the child with the highest value.

Since generating the full game tree is often infeasible, the search is typically cut off at a certain depth. The leaf nodes are then scored using an **evaluation function** that estimates the desirability of the position. It's best to stop the search only at "quiescent" positions to avoid the **horizon effect**, where a critical event is pushed just beyond the search depth limit.

### 5.2 Alpha-Beta Pruning

**Alpha-Beta pruning** is an optimization of the minimax algorithm that provides the same result while pruning large parts of the search tree. It eliminates branches that cannot possibly influence the final decision. It maintains two values during the search:

- **Alpha ($\alpha$):** The best (highest) value found so far for MAX on the path to the root.
- **Beta ($\beta$):** The best (lowest) value found so far for MIN on the path to the root.
- Prune branches below MIN node where $\beta \leq \alpha$ of its MAX-predecessor. Prune branches below MAX node where $\alpha \geq \beta$ of its MIN-predecessor.

The efficiency of alpha-beta pruning is highly dependent on move ordering. In the best case, it can reduce the effective branching factor from $b$ to $\sqrt{b}$, allowing the search to go twice as deep in the same amount of time.

### 5.3 Games with an Element of Chance

Games like Backgammon involve randomness. To handle this, the standard game tree is augmented with **chance nodes**.

- These nodes are inserted between the regular MAX and MIN nodes and represent random events, like a dice roll.
- Instead of a min or max operation, the value of a chance node is the **expected value** of its children, calculated by summing the values of all possible outcomes weighted by their probabilities.
- This significantly increases the branching factor and complexity of the search.

# 6 Propositional Logic

Rational agents often need to represent knowledge about their world to make logical deductions. Propositional logic is a formal language for representing and reasoning with this knowledge.

## 6.1 Knowledge-Based Agents

A knowledge-based agent uses a **knowledge base (KB)**, which is a set of sentences expressed in a formal language. The agent can interact with the KB via:

- **TELL(KB,$\alpha$)=KB'** the KB new information it perceives from the environment.
- **ASK(KB,$\alpha$)=yes** the KB what action to take, which is answered by inferring new knowledge from the KB.

It is crucial to distinguish between the **syntax** of the language (the structure of its sentences) and its **semantics** (the meaning or truth of those sentences).

## 6.2 Syntax and Semantics of Propositional Logic

- **Syntax:** A **literal** is an atomic proposition or its negation. A **clause** is a disjunction of literals.
- **Semantics:** The truth of a formula is determined by an **interpretation** (or truth assignment), which assigns a truth value (True or False) to each atomic proposition. An interpretation that makes a formula true is called a **model** of that formula.

A formula can be **satisfiable** (has at least one model), **unsatisfiable** (has no models), **falsifiable** (a model exists that does not satisfy the formula) or **valid** (a tautology, true under all interpretations).

## 6.3 Logical Entailment and Inference

- **Logical Entailment** ($\text{KB} \models \alpha$): A sentence $\alpha$ is logically entailed by a knowledge base KB if $\alpha$ is true in all models of KB. This means that $M(\text{KB}) \subseteq M(\alpha)$, where $M(\phi)$ is the set of all models for a formula $\phi$.
- **Inference** ($\text{KB} \vdash_i \alpha$): Inference is the process of deriving new sentences from existing ones using syntactic rules. A good inference procedure should be:
  - **Sound:** It only derives sentences that are logically entailed.
  - **Complete:** It can derive all sentences that are logically entailed.

## 6.4 CNF and DNF

For every formula, there exists at least one equivalent formula in CNF and one in DNF. A formula in DNF is satisfiable *iff* one disjunct is satisfiable.

$$\text{CNF} : \bigwedge_{i=1}^{n} \left( \bigvee_{j=1}^{m_i} l_{i,j} \right) \quad ; \quad \text{DNF} : \bigvee_{i=1}^{n} \left( \bigwedge_{j=1}^{m_i} l_{i,j} \right)$$

A formula can be turned into CNF by following the steps:

1. Eliminate $\Rightarrow$ and $\Leftrightarrow$ like so: $\alpha \Rightarrow \beta \rightarrow (\neg\alpha \vee \beta)$
2. Move $\neg$ inwards: $\neg(\alpha \wedge \beta) \rightarrow (\neg\alpha \vee \neg\beta)$ (De Morgan's laws)
3. Distribute $\vee$ over $\wedge$ like so: $((\alpha \wedge \beta) \vee \gamma) \rightarrow (\alpha \vee \beta) \wedge (\beta \vee \gamma)$
4. Simplify: $\alpha \vee \alpha \rightarrow \alpha$ etc.

### 6.5 Resolution

Resolution is a sound and refutation-complete inference rule for deriving new formulae from a KB, that does not depend on testing every interpretation (logical implication).

- **Goal:** To prove KB $\models \alpha$, the resolution algorithm proves that the set of sentences KB $\cup \{\neg\alpha\}$ is unsatisfiable. This relies on the contradiction theorem.
- **Requirement:** All sentences must be converted to CNF, which is a conjunction of clauses (disjunctions of literals). Equivalently, we can turn the KB sentences into a set of clauses:

$$\{(P \vee Q) \wedge (R \vee \neg P) \wedge S\} \rightarrow \{\{P, Q\}, \{R, \neg P\}, \{S\}\}$$

- **The Resolution Rule:** From two clauses containing a complementary literal (e.g., $l$ and $\neg l$), a new clause called the **resolvent** is derived. The resolvent contains all literals from the original two clauses except for the complementary pair.

$$\frac{C_1 \cup \{l\}, \quad C_2 \cup \{\neg l\}}{C_1 \cup C_2}$$

- **Proof:** The algorithm repeatedly applies the resolution rule. If the **empty clause** ($\square$) is derived, it signifies a contradiction, proving that the original set of clauses was unsatisfiable and thus that $KB \models \alpha$.

**Left out:** Special symbols, Interpretation, Implication theorems, Action selection

## 7 Satisfiability and Model Construction

The Boolean Satisfiability Problem (SAT) is a foundational problem in computer science. Given a propositional formula in CNF, the goal is to find a satisfying assignment of truth values (a model) or prove that none exists. Modern SAT solvers provide a highly efficient way to solve many practical NP-hard problems, including hardware/software verification and CSPs.
SATs can be formulated as CSPs, where the CSP variables are the symbols of the alphabet, the domain of values is $\{T, F\}$, and the constraints are given by clauses.

### 7.1 The DPLL Algorithm

The **Davis-Putnam-Logemann-Loveland (DPLL)** algorithm is a complete and correct backtracking search procedure for solving SAT. It improves upon simple backtracking by using powerful inference rules to prune the search space.

**Definition 4:**

Given a set of clauses $\Delta$ defined over a set of variables $\Sigma$, return "satisfiable" if $\Delta$ is satisfiable, otherwise return "unsatisfiable".

1. If $\Delta = \emptyset$ return "satisfiable"
2. If $\square \in \Delta$ return "unsatisfiable" ($\square$ being the empty clause)
3. **Unit-propagation rule:** If $\Delta$ contains a unit-clause $C$, that is an unassigned literal, assign a truth-value to the variable in $C$ that satisfies $C$, simplify $\Delta$ to $\Delta'$ and return DPLL($\Delta'$).
4. **Splitting rule:** When no further progress can be made by unit propagation, select an unassigned variable $v$ from $\Sigma$ and recursively try assigning it true and then false. After assigning each truth value, simplify $\Delta$ to $\Delta'$ and call DPLL($\Delta'$).

### 7.2 Conflict-Driven Clause Learning (CDCL)

Modern SAT solvers are based on an enhancement of DPLL called **Conflict-Driven Clause Learning (CDCL)**. While DPLL is effective, it "forgets" the reason for a conflict when it backtracks. CDCL improves upon this by learning from its mistakes. One of its core features is **Conflict Analysis and Clause Learning:** When a conflict occurs, the solver analyzes the chain of implications that led to it and generates a new **conflict clause**. This new clause is added to the knowledge base and helps prune the search by preventing the same conflict from recurring.

**Definition 5:**

The CDCL algorithm works as follows:

1. **Unit-propagation rule:** If $\Delta$ contains a unit clause $C$ for $L$, extend the trail with $L^C$.
2. **OR:** If there is a conflict, call ANALYZE(conflict):
   (a) if the learned clause is empty, return "unsatisfiable"
   (b) otherwise, backjump: Learn the clause and unit-propagate to the earliest possible split, undoing all splits done later
3. **Splitting rule:** Select from $\Sigma$ an unassigned variable $v$ and assign it a truth value. Extend the trail with $L^\dagger$.
4. If there is no unassigned literal, return "satisfiable(trail)"
5. Go to step 1

## 8 Predicate Logic (PL1)

While propositional logic is useful, it cannot represent the internal structure of statements or relationships between objects. For example, from "All men are mortal" and "Socrates is a man," it cannot conclude that "Socrates is mortal." First-Order Predicate Logic (PL1) is a more expressive language that overcomes these limitations.

### 8.1 Syntax

PL1 introduces several new syntactic elements:

- Quantifiers ($\forall$, $\exists$), equality ($=$), brackets, variables $(x, x_1, ..., x', x'', ..., y, z)$
- **Function symbols** (e.g. weight(), color()), lowercase. 0-ary functions are constants.
- **Predicate symbols** (e.g. Red(), Block()), uppercase. 0-ary predicates are prop. logic atoms.
- **Terms:** Expressions that represent objects. A term can be a constant, a variable, or a function applied to other terms (e.g., $a, x, f(x)$). Terms without variables are called **ground terms**.
- **Atomic formulae:** Represent statements about objects. If $t_1, ..., t_n$ are terms and $P$ is an $n$-ary predicate, then $P(t_1, ..., t_n)$ is an atomic formula, so would be $t_1 = t_2$.
- **Formulae:** If $\phi$ and $\psi$ are formluae and $x$ ar variable, then every operator application (e.g. $\neg\phi$, $\phi \wedge \psi$, $\exists x\psi$) are formulae. Quantifiers are as strongly binding as $\neg$.

Formulae with no free variables are called closed formulae or sentences. With closed formulae, variable assignments can be ignored.

### 8.2 Semantics:

The meaning of a PL1 formula is given by an **interpretation**, which consists of:

- A non-empty **domain** $D$ of objects.
- A mapping that assigns objects in $D$ to constant symbols, functions over $D$ to function symbols, and relations over $D$ to predicate symbols.

The truth of a formula is evaluated relative to an interpretation and a variable assignment.

### 8.3 Reduction to Propositional Logic

Logical entailment in full PL1 is undecidable, meaning no algorithm exists that can decide for all formulae whether one is entailed by a knowledge base. However, for the special case of a **finite domain**, PL1 can be reduced to propositional logic through a process called **propositionalization**. This is achieved by first assuming a **Domain Closure Axiom**, which states that the only objects in the domain are those named by the constants. Quantifiers are then eliminated through **instantiation**:

- A universally quantified formula is replaced by a conjunction of the formula with the variable instantiated for every constant in the domain.

$$\forall x\varphi \quad \longrightarrow \quad \varphi[x/c_1] \wedge \varphi[x/c_2] \wedge \dots$$

- An existentially quantified formula is replaced by a disjunction.

$$\exists x\varphi \quad \longrightarrow \quad \varphi[x/c_1] \vee \varphi[x/c_2] \vee \dots$$

**Notation:** If $\varphi$ is a formula, then $\varphi[x/a]$ is the formula with all free occurences of $x$ replaced by $a$.

This process results in a (potentially very large) propositional theory that is equivalent to the original PL1 theory and can be solved using standard SAT solvers.

## 9 Action Planning

Planning is the process of finding a sequence of actions to achieve a goal. In AI, this involves declaratively specifying a problem (using logic) — describing the initial state, the available actions, and the goal — and using a domain-independent planner to automatically find a solution.

### 9.1 Planning Formalisms

To describe planning problems in a domain-independent way, formal languages are used.

#### 9.1.1 STRIPS (STanford Research Institute Problem Solver):

- **States** $\mathcal{S}$ are represented as sets of true propositions (ground atoms), assuming a **Closed World Assumption** (any atom not listed is false).
- **Actions** (or operators) **O** are defined by:
  - **Preconditions:** A set of propositions that must be true for the action to be executable.
  - **Effects:** A set of propositions that describe how the state changes. This is split into an **add list** (atoms that become true) and a **delete list** (atoms that become false).
- A **plan** $\Delta$ is a sequence of actions that transforms the initial state **I** into a state that satisfies the goal conditions **G**. A **planning task** is a 4-tuple $\langle \mathcal{S}, \mathbf{O}, \mathbf{I}.\mathbf{G} \rangle$.

### 9.1.2 PDDL (Planning Domain Description Language):

The de facto standard language for planning problems. It extends STRIPS with more expressive features like typing, conditional effects, and numerical resources.

## 9.2 Basic Planning Algorithms

Planning can be framed as a search problem on a state-transition graph. Here, nodes are defined by value assignments to states and labeled edges are the actions that change the appropriate nodes.

- **Progression Planning (Forward Search):** This approach searches forward from the initial state towards a goal state. At each step, it considers all applicable actions and generates successor states. This is intuitive and easy to implement.
- **Regression Planning (Backward Search):** This approach searches backward from the goal description towards the initial state. It starts with the goal and determines what conditions must have been true before an action was taken to achieve it. This can be more efficient as it only considers actions relevant to the goal.

# 10 Making Decisions Under Uncertainty

Agents in the real world must often act with incomplete or uncertain information. Logical rules can be brittle, as it's impossible to list all preconditions for an action (the qualification problem) or all possible causes for an effect. Probability theory provides a formal framework for representing and reasoning with an agent's degree of belief in the face of uncertainty. When combined with utility theory, it forms the basis of **decision theory**, which allows an agent to choose actions that maximize its expected utility.

## 10.1 Foundations of Probability Theory

Probability is governed by a set of axioms, from which all other properties are derived.

- An **unconditional (or prior) probability** $P(A)$ represents the degree of belief in a proposition $A$ in the absence of other information.
- A **conditional (or posterior) probability** $P(A|B)$ represents the belief in $A$ given that evidence $B$ is known.
- The **Product Rule** states: $P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$.
- **Independence:** $a$ and $b$ are independent *iff* $P(a|b) = P(a)$ ($\Rightarrow P(a \wedge b) = P(a)P(b)$)
- **Bayes' Rule** is a cornerstone of probabilistic reasoning, allowing us to update beliefs based on new evidence. It relates causal and diagnostic knowledge:

$$P(\text{cause}|\text{effect}) = \frac{P(\text{effect}|\text{cause})P(\text{cause})}{P(\text{effect})}$$

## 10.2 Probabilistic Inference

The **joint probability distribution** specifies a probability for every possible state of the world (i.e., every complete assignment of values to all random variables, like in a probability table). In a probability table the sum of all fields is 1 (disjunction of events). The probability for any random variable can be computed from the joint distribution through **marginalization** (summing out variables over rows or columns). Conditional probabilities can be obtained using marginal probabilities.
To make probabilistic inference feasible, we rely on **conditional independence**. Two variables $A$ and $B$ are conditionally independent given $C$ if $P(A|B,C) = P(A|C)$. This means that once we know the outcome of $C$, learning about $B$ provides no additional information about $A$.

### 10.3 Bayesian Networks

A **Bayesian Network** is a powerful tool for representing and reasoning with uncertain knowledge. It provides a compact representation of the joint probability distribution by making explicit the conditional independence relationships in a domain.

- **Structure:** A network is a **Directed Acyclic Graph (DAG)** where nodes represent random variables and directed edges represent direct causal influences.
- **Parameters:** Each node has an associated **Conditional Probability Table (CPT)** that quantifies the probability of that node's value given the values of its parent nodes.
- **Semantics:** A Bayesian network encodes the assumption that each node is conditionally independent of its non-descendants, given its parents. This allows the full joint probability to be factored into a product of local conditional probabilities:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(x_i)) \quad ,$$

  where $X_1, ..., X_n$ are the topologically ordered nodes and $x_1, ..., x_n$ are the values of the variables.
- **Inference:** The primary task is to compute the posterior probability of a query variable given some observed evidence $e$. The network gives a complete representation of the full joint distribution. A query can be answered by computing sums of products of conditional probabilities (summing over the hidden variables). Exact inference is generally NP-hard (approximation via MCMC). Worst-case time complexity is $O(2^n)$ for $n$ boolean variables. For singly connected graphs and polytrees, time and space complexity is linear in the number of nodes.

**Left out:** ENUMERATION-ASK, Bayesian updating

## 11 Acting Under Uncertainty

When an agent operates in a stochastic environment, it must consider the uncertainty of action outcomes. Decision theory, which combines utility theory and probability theory, provides a framework for making rational choices by selecting actions that maximize the agent's expected utility.

### 11.1 Utility Theory and the MEU Principle

- A **utility function** $U(S)$ assigns a numerical value to each state $S$, representing its desirability to the agent.
- The axioms of utility theory (e.g., Orderability, Transitivity, Continuity) provide a foundation for rational preferences. If an agent's preferences conform to these axioms, a utility function that represents those preferences is guaranteed to exist.
- The **Principle of Maximum Expected Utility (MEU)** states that a rational agent should choose the action that maximizes its expected utility. The expected utility of an action $A$ given evidence $E$ is calculated as:

$$EU(A|E) = \sum_i P(\text{Result}(A) = i|\text{Do}(A), E)U(i)$$

  Problem: The probability part of MEU requires a complete causal model of the world (e.g. requiring constant belief network updating, NP-hard), while the utility part of MEU requires search or planning.

## 11.2 Sequential Decision Problems and MDPs

Many problems require an agent to make a sequence of decisions over time. A **Markov Decision Process (MDP)** is a formal model for sequential decision-making in a fully observable, stochastic environment. An MDP is defined by:

- A set of states $S$.
- A set of actions $A$.
- A **transition model** $P(s'|s,a)$, giving the probability of reaching state $s'$ from state $s$ after taking action $a$.
- A **reward function** $R(s)$, specifying the immediate reward received in state $s$.

A **policy**, $\pi(s)$, is a mapping from states to actions. The goal is to find the **optimal policy**, $\pi^*$, which maximizes the long-term expected reward. For finite horizon problems, $\pi*$ depends on current state and remaining steps to go (nonstationary). For inifite horizon problems, it only depends on the current state (stationary).

## 11.3 The Bellman Equation and Algorithms for MDPs

The long-term utility of a state, $U(s)$, is the expected sum of future discounted rewards, using a discount factor $\gamma \in [0,1)$. The relationship between the utility of a state and its successors is defined by the **Bellman Equation**:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

This equation is the foundation for algorithms that solve MDPs. The agent simply chooses the action that maximizes the expected utility of the subsequent state:

$$\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a)U(s')$$

- **Value Iteration:** An iterative algorithm that calculates the optimal utility for each state. It starts with arbitrary utilities and repeatedly applies the Bellman update until the values converge. Bellman update:

$$U'(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

- **Policy Iteration:** An alternative algorithm that often converges faster. It alternates between two steps until the policy becomes stable: (1) **Policy Evaluation**, where given a policy $\pi_t$ it calculates the utilities for the current policy $U_t = U^{\pi_t}$, and (2) **Policy Improvement**, where it uses these utilities to create a better policy according to:

$$\pi_{t+1}(s) = \arg\max_a \sum_{s'} P(s'|s,a)U_t(s')$$

**Left out:** Concrete utility theory axioms

# 12 Machine Learning

An agent is said to learn if it improves its performance on a task with experience. Machine learning provides the tools for agents to adapt and improve automatically, rather than being explicitly programmed for every possible scenario. A learning agent is typically composed of a performance element (which acts), a learning element (which improves), a critic (which provides feedback), and a problem generator (which suggests exploratory actions).

## 12.1 Types of Learning

Machine learning is often categorized by the type of feedback the agent receives.

- **Supervised Learning:** The agent learns a function from a training set of labeled input-output pairs $\{(x_1, y_1), \ldots, (x_N, y_N)\}$. The goal is to find a hypothesis $h$ that can generalize from this data to accurately predict the output for unseen inputs.
  A key principle in supervised learning is **Ockham's razor**, which advocates for choosing the simplest hypothesis that is consistent with the data.
- **Reinforcement Learning:** The agent learns from rewards or punishments (reinforcement signals). It is not told which action to take but must discover a policy that maximizes its cumulative reward over time through trial and error.
- **Unsupervised Learning:** The agent learns patterns directly from unlabeled data, without any explicit feedback.

## 12.2 Decision Trees

A decision tree is a simple yet powerful model used for classification and regression. It represents a function that takes a set of input attributes and returns a decision.

- **Structure:** The tree consists of internal nodes that test an attribute, branches corresponding to the attribute's values, and leaf nodes that specify the final output or classification.
- **Learning Algorithm:** Finding the smallest possible decision tree is an intractable problem. Instead, a greedy, divide-and-conquer algorithm is used to build the tree:
  1. If all examples in the current subset have the same classification, create a leaf node.
  2. Otherwise, select the "best" attribute to split the examples into smaller subsets.
  3. Recurse on each subset.
- **Attribute Selection:** To choose the "best" attribute for a split, the algorithm calculates the information gain for each attribute. Information gain measures the reduction in uncertainty, or **entropy**, after the split. The attribute with the highest information gain is chosen. Entropy is defined as:
  $$I(P(y_1), \ldots, P(y_n)) = \sum_{i=1}^{n} -P(y_i) \log_2 P(y_i)$$

## 12.3 Assessing Performance

To evaluate a learning algorithm, it is crucial to separate the data into a **training set** and a **test set**. The training set is used to learn the hypothesis (e.g., build the decision tree), while the test set, which the model has never seen, is used to evaluate its performance and generalization ability.
A **learning curve** can be plotted to show how the model's performance on the test set improves as the size of the training set increases.

# 13 Deep Learning

Deep learning is a subfield of machine learning that has achieved state-of-the-art results across a wide range of tasks, including computer vision, speech recognition, natural language processing, and playing complex games like Go.

## 13.1 Representation Learning

Traditional machine learning pipelines often require extensive manual **feature engineering** to transform raw data (like images or text) into a suitable representation from which a model can learn.

- **Representation Learning** refers to a set of methods that automatically discover the necessary features or representations from raw data.
- **Deep Learning** is a form of representation learning that uses models with multiple layers to learn a **hierarchy of features**. Each layer transforms the input from the previous layer into a more abstract and complex representation.

## 13.2 Multilayer Perceptrons (MLPs)

- **Structure:** An MLP consists of an input layer, one or more **hidden layers**, and an output layer. Each layer contains interconnected nodes called neurons.
- **Computation:** The output of each neuron is calculated by taking a weighted sum of its inputs, adding a bias, and then passing the result through a non-linear **activation function**. Common activation functions include the logistic sigmoid, tanh and ReLU.
- **Training:** The network's parameters (weights and biases) are learned by minimizing a loss function that measures the discrepancy between the network's predictions and the true labels. This is typically done using **Stochastic Gradient Descent (SGD)**. The gradients required for this optimization are calculated efficiently using the **backpropagation** algorithm.

## 13.3 Advanced Architectures

- **Convolutional Neural Networks (CNNs):** CNNs are specialized for processing grid-like data, such as images. They use a mathematical operation called **convolution**, where a small filter (or kernel) is slid across the input. This allows the network to efficiently learn a hierarchy of local patterns (e.g., edges, textures, shapes) while sharing parameters, making them highly effective for computer vision tasks.
- **Recurrent Neural Networks (RNNs):** RNNs are designed to handle sequential data, like text or time series. They have loops in their structure that create an internal state or "memory," allowing them to process sequences of arbitrary length. However, they can be difficult to train on long sequences due to issues like vanishing or exploding gradients.
- **Transformers:** A more modern architecture that has become dominant in natural language processing. Instead of processing sequences step-by-step like RNNs, Transformers use an **attention mechanism** to weigh the importance of all input elements simultaneously. This allows them to capture long-range dependencies in data much more effectively.