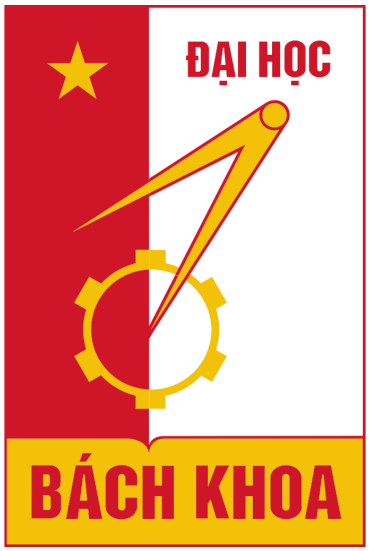
Hanoi University of Science and Technology
School of Information and Communication Technology



IT3160E - Introduction to Artificial Intelligent

Student Name: Nguyen Duc Anh
Student ID: 20235890

Chapter 1: Introduction

1 What is Artificial Intelligence (AI)?

Core Idea: AI is about making computers "smart" – able to do things that normally require human intelligence. It's a field of both science (understanding how intelligence works) and engineering (building intelligent systems).

Four Views of AI These views are important for understanding the goals of AI:

- **Thinking like humans:** Trying to make computers think the way we do (like having consciousness, emotions – this is very hard and mostly philosophical).
- **Thinking rationally:** Making computers use logic and perfect reasoning (like a perfect chess player – always making the best move).
- **Acting like humans:** Making computers behave in ways that seem intelligent to us (like passing a conversation test – the Turing Test).
- **Acting rationally:** Making computers do the "right thing" to achieve a goal, given what they know (even if it doesn't perfectly mimic humans). This is the most practical and common view in modern AI.

"Right thing" means: Maximize their performance, given available information.
Key Takeaway: Most modern AI focuses on acting rationally – achieving goals effectively, not necessarily perfectly replicating human thought.

2 The Rational Agent Concept (Crucial!)

Agent: Anything that can perceive its environment (using sensors) and act upon it (using actuators). Think of a robot vacuum cleaner: it senses dirt and moves to clean it. A software agent might sense website clicks and recommend products.
Rational Agent: An agent that acts in a way that is expected to maximize its "success" or "performance" based on what it knows. This is the central concept.

Key Definitions:

- **Performance Measure:** The way we define "success" for an agent. For the vacuum, it might be "amount of dirt cleaned." For a chess program, it's "winning the game."
- **Percept:** What the agent senses at any given moment (e.g., the vacuum sees a dirty spot).
- **Percept History:** The entire sequence of everything the agent has ever sensed.
- **Agent Function:** Mathematically, $f : P^* \to A$, where:
  - f: function,
  - P^*: perception history,
  - A: action.

  This function defines the agent's behavior. Finding/creating this function is the core problem in AI.

Important Note: Perfect rationality is often impossible because of limited computing power and time. Real-world AI has to be *practically rational* – good enough, given the constraints.

3 AI, Machine Learning, Deep Learning, and Industry 4.0

- **Artificial Intelligence (AI):** The broad field of creating intelligent systems.
- **Machine Learning (ML):** A subset of AI where systems learn from data without being explicitly programmed for every situation. Instead of writing rules, you feed the system data, and it figures out the rules itself.

- **Deep Learning (DL):** A subset of ML that uses artificial neural networks with many layers ("deep" networks) to learn complex patterns from data. It's been very successful in image recognition, natural language processing, and other areas.
- **Data Science (DS):** A broader field that includes AI and ML, but also focuses on extracting knowledge and insights from data (including data analysis, statistics, visualization, etc.).
- **Industry 4.0:** The current trend of automation and data exchange in manufacturing technologies, heavily driven by AI and ML.

4 Related Fields (AI Draws From Many Disciplines)

AI is interdisciplinary, drawing from the following fields:

- **Philosophy:** Logic, reasoning, the nature of knowledge, and the mind.
- **Mathematics:** Formal logic, algorithms, computation, probability, and statistics.
- **Economics:** Decision-making, utility (measuring the value of outcomes), and game theory.
- **Neuroscience:** How the brain works, providing inspiration for artificial neural networks.
- **Psychology:** How humans think and behave, perception, and learning.
- **Computer Engineering:** Building the hardware and software that AI runs on.
- **Control Theory:** Designing systems that achieve goals in a stable and optimal way.
- **Linguistics:** The structure of language, crucial for natural language processing.

5 Main Subfields of AI (Areas of Specialization)

AI encompasses several specialized areas:

- **Search & Planning:** Finding sequences of actions to achieve goals (like a robot navigating a maze).
- **Knowledge Representation & Reasoning:** Storing and using knowledge logically (like building a system that can answer medical questions).
- **Machine Learning:** Learning from data (as described above).
- **Machine Perception:** Giving computers the ability to "see" (computer vision) and "hear" (speech recognition), and understand natural language.
- **Robotics:** Building physical robots that can interact with the world.
- **Multi-agent Systems:** Designing systems where multiple AI agents interact (like self-driving cars coordinating traffic).

6 Libraries and Frameworks

6.1 TensorFlow (Framework)

Developed by: Google Brain
Key Features:

- *Computation Graph:* Defines AI models as a graph of mathematical operations. This allows for efficient execution, especially on GPUs (Graphics Processing Units, which are very good at parallel processing).
- *Automatic Differentiation:* Automatically calculates the gradients (derivatives) needed for training neural networks. This is crucial for optimizing models.

- *Scalability:* Can handle everything from small experiments on a laptop to large-scale deployments on clusters of servers.
- *TensorBoard:* A visualization toolkit for monitoring training progress and understanding model behavior.
- *Keras Integration:* Keras (see below) is now tightly integrated with TensorFlow, providing a high-level, user-friendly API.
- *TensorFlow Lite:* For deploying models on mobile and embedded devices.
- *TensorFlow.js:* For running models in web browsers.

Best For: Deep learning, especially large-scale models, research, production deployments.

6.2 PyTorch (Framework)

Developed by: Facebook's AI Research lab (FAIR) / Meta
Key Features:

- *Dynamic Computation Graph:* Unlike TensorFlow's static graph, PyTorch's graph is built dynamically as the code runs. This makes it more flexible and easier to debug.
- *Python-First Approach:* Feels more natural to Python programmers, with a more intuitive API.
- *Strong Community and Research Focus:* Very popular in the research community.
- *Eager Execution:* Allows you to run operations immediately, making debugging and experimentation easier.
- *Good GPU Support:* Excellent performance on GPUs.

Best For: Deep learning, research, prototyping, projects where flexibility and ease of debugging are important.

6.3 Keras (High-Level Framework)

Developed by: François Chollet (originally independent, now part of TensorFlow)
Key Features:

- *User-Friendliness:* Designed to be easy to learn and use, with a simple, high-level API.
- *Modularity:* Models are built by connecting layers and components like Lego bricks.
- *Backend Flexibility:* Can run on top of TensorFlow, Theano (older, less used now), or CNTK. (TensorFlow is the most common backend.)
- *Good for Beginners:* Great for learning the basics of deep learning.

Best For: Rapid prototyping, learning deep learning, building models without needing to deal with low-level details. Often used as the "entry point" to TensorFlow.

6.4 Scikit-learn (Library)

Developed by: Community-driven (open source)
Key Features:

- *Traditional Machine Learning:* Focuses on classic machine learning algorithms (not deep learning). Includes:
  - Classification: (e.g., identifying spam emails)
  - Regression: (e.g., predicting house prices)
  - Clustering: (e.g., grouping customers with similar behavior)
  - Dimensionality Reduction: (e.g., simplifying data while preserving important information)

  - Model Selection: (e.g., choosing the best algorithm and parameters)
  - Preprocessing: (e.g., cleaning and transforming data)
- *Simple and Consistent API:* Easy to learn and use.
- *Well-Documented:* Excellent documentation and tutorials.
- *Built on NumPy, SciPy, and Matplotlib:* Integrates well with other scientific Python libraries.

Best For: General-purpose machine learning tasks (excluding deep learning), data analysis, prototyping, education.

6.5 NLTK (Natural Language Toolkit) (Library)

Developed by: Steven Bird, Ewan Klein, and Edward Loper
Key Features:

- *Text Processing:* Provides tools for tasks like:
  - Tokenization (splitting text into words and sentences).
  - Stemming and Lemmatization (reducing words to their root forms).
  - Part-of-Speech Tagging (identifying nouns, verbs, adjectives, etc.).
  - Parsing (analyzing the grammatical structure of sentences).
  - Named Entity Recognition (identifying people, places, organizations, etc.).
  - Sentiment Analysis (determining the emotional tone of text).
- *Corpora and Lexical Resources:* Includes access to a large collection of text datasets and word lists (like WordNet).
- *Educational Focus:* Often used for teaching and learning NLP.

Best For: Natural language pre-processing, analysis tasks.

Chapter 2: Intelligent Agent

1 What is an Agent?

Definition: An agent is anything that can perceive its environment (through sensors) and act upon that environment (through actuators). It's a general concept – humans are agents, robots are agents, even a simple thermostat is an agent.
Examples:

- *Human Agent:*
  - Sensors: eyes, ears, skin, etc.
  - Actuators: hands, legs, mouth.
- *Robot Agent:*
  - Sensors: cameras, microphones, touch sensors.
  - Actuators: wheels, arms, grippers, speakers.
- *Software Agent:*
  - Sensors: keyboard input, network data, file contents.
  - Actuators: displaying text, sending network requests, writing to files.

2 Agent Function and Agent Program

Agent Function ($f : P^* \rightarrow A$): This is the ideal behavior of the agent. It's a mathematical function that says, "Given the entire history of everything the agent has ever perceived (P^*), what action (A) should it take?" This function describes what the agent should do, not how it does it.

- f: function
- P^*: perception history
- A: action

Agent Program: This is the actual implementation of the agent function (or an approximation of it). It's the code that runs on the agent's architecture (computer, robot, etc.).
Agent = Architecture + Program: The physical "body" (architecture) plus the "mind" (program) make up the complete agent.

3 Rationality

Rational Agent: An agent that tries to do the "right thing" – to act in a way that maximizes its performance measure.
Performance Measure: How we define "success" for the agent. It's what the agent is trying to optimize. Examples:

- Vacuum cleaner: Amount of dirt cleaned, time taken, energy used.
- Chess player: Winning the game.
- Self-driving car: Reaching the destination safely and efficiently.

Key Idea: A rational agent chooses actions based on its percept history and its built-in knowledge to maximize its expected performance.
Not Omniscience: Rationality doesn't mean knowing everything. An agent can only act based on what it has perceived. It might make mistakes due to incomplete information.
Autonomy: Learn and adapt.

4 PEAS: Describing the Task Environment

PEAS is an acronym to help define the problem an agent is designed to solve:

- **Performance Measure:** How do we judge success?
- **Environment:** What is the agent's surroundings?
- **Actuators:** How does the agent affect the environment?
- **Sensors:** How does the agent perceive the environment?

Example: Taxi-driving agent

- P: Safe, fast, legal, comfortable trip, maximize profit.
- E: Roads, traffic, pedestrians, customers.
- A: Steering wheel, accelerator, brake, signal, horn.
- S: Cameras, speedometer, GPS, accelerometer, microphone.

5 Environment Types (Important for Agent Design)

The type of environment greatly influences how we design an agent. Here are the key distinctions:

5.1 Fully Observable vs. Partially Observable

- **Fully:** The agent's sensors can see the entire state of the environment at any time (e.g., a chessboard).
- **Partially:** The agent can only see parts of the environment (e.g., a self-driving car can't see everything around it).

5.2 Deterministic vs. Stochastic

- **Deterministic:** The next state of the environment is completely determined by the current state and the agent's action (e.g., a chess game – the rules are fixed).
- **Stochastic:** There's randomness involved; the next state isn't perfectly predictable (e.g., rolling dice, the stock market).
- **Strategic:** Deterministic, except for the actions of other agents.

5.3 Episodic vs. Sequential

- **Episodic:** The agent's experience is divided into independent "episodes." Each episode is a perceive-then-act sequence, and the choice of action only depends on that episode (e.g., classifying images one at a time).
- **Sequential:** The current decision can affect future decisions (e.g., playing chess – each move affects the future game state).

5.4 Static vs. Dynamic

- **Static:** The environment doesn't change while the agent is "thinking" (deliberating).
- **Dynamic:** The environment can change while the agent is deliberating. The agent might need to hurry up!
- **Semi-dynamic:** Environment is unchanged while the agent is deliberating, but the performance score does change.

5.5 Discrete vs. Continuous

- **Discrete:** There's a finite (or countable) number of distinct states, percepts, and actions (e.g., a chess game has a finite number of board positions).
- **Continuous:** States, percepts, and actions can vary smoothly over a range (e.g., the steering angle of a car).

5.6 Single-agent vs. Multi-agent

- **Single:** The agent is the only one acting in the environment.
- **Multi-agent:** Multiple agents are interacting (could be cooperative or competitive).

6 Agent Types (Increasing Complexity)

These are broad categories of how agents can be designed:

6.1 Simple Reflex Agents

- React directly to the current percept, ignoring the percept history.
- Use condition-action rules (if-then rules): "If you see a red light, stop."
- Simple, but limited – can't handle partially observable environments well.

6.2 Model-Based Reflex Agents

- Keep track of an internal state that represents their belief about the world, even parts they can't directly see.
- Use a model of how the world works to update this internal state.
- Still use condition-action rules, but the conditions can refer to the internal state.
- Better than simple reflex agents in partially observable environments.

6.3 Goal-Based Agents

- Have a goal – a description of desirable situations.
- Reason about how their actions will affect the world and whether they will achieve the goal.
- More flexible than reflex agents – can handle new situations if they can figure out how to achieve the goal.
- Often involve search and planning.

6.4 Utility-Based Agents

- Have a utility function that measures how "good" a state is (not just whether it achieves a goal).
- Try to maximize their expected utility.
- Useful when there are multiple possible goals, or when there's uncertainty about achieving goals. They can make trade-offs.

6.5 Learning Agents

- Learning allows the agent to improve their performance over time.
- **The Four Elements of a Learning Agent:**
  - *Performance Element:* Undertakes the choice of action based on current knowledge and perceptions.
  - *Critic:* Evaluates the performance, providing feedback on how well the agent's actions align with its goals.
  - *Learning Element:* Helps to improve the performance – based on the critic's feedback, it modifies (improves) the performance element.
  - *Problem Generator:* Helps to generate new experiences, suggesting actions that allow the agent to explore and learn from new situations.

7 Multi-Agent Systems (MAS)

Definition: Systems involving multiple interacting agents, each with its own perceptions, actions, and potentially goals.
Why MAS? Used for distributed problems, multiple stakeholders, increased complexity management, and system robustness.

7.1 Environment Types

- **Collaborative:** Agents share common goals, requiring coordination, communication, and task allocation.
  - Example: Robot teams.
- **Competitive:** Agents have conflicting goals, necessitating strategic reasoning, opponent modeling, and potentially deception.
  - Example: Games, auctions.
- **Mixed:** Environment contains both collaborative and competitive elements.

7.2 Key Challenges & Theories

- **Communication:** Agents need languages and protocols to exchange information.
- **Coordination:** Agents must work together effectively (centralized or decentralized).
- **Game Theory:** Crucial for understanding competitive interactions (e.g., Nash Equilibrium, Dominant Strategy).
- **Opponent/Collaborative Modeling:** Predicting the actions of other agents.
- **Negotiation:** Agents bargain to reach agreements.

7.3 Example Application

Multi-Agent Pathfinding (MAPF): Finding collision-free paths for multiple agents.

Chapter 3: Uninformed Searching

1 Core Concepts

- **Problem Solving as Search:** The fundamental idea is to frame problem-solving as finding a sequence of actions that transform an initial state into a goal state.
- **Agent:** An entity that perceives its environment and acts upon it. In this context, the agent is trying to find a solution.
- **State:** A representation of the world at a particular point in time. States can be abstract (representing a set of real-world states) to simplify the problem.
- **Action:** Something the agent can do to change the state of the world.
- **State Space:** The set of all possible states reachable from the initial state by any sequence of actions. Often visualized as a graph.
- **State-Transition Function (Successor Function):** $S(\text{current\_state})$ returns a set of $\langle \text{action}, \text{next\_state} \rangle$ pairs, defining how actions change the state. Example: $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$.
- **Goal Test:** A function that determines whether a given state is a goal state. Can be direct (e.g., state = "Bucharest") or indirect (e.g., Checkmate(state)).
- **Path Cost:** A numerical cost associated with a sequence of actions (a path). Often additive (the sum of the costs of individual actions). $c(x, a, y)$ represents the cost of taking action a from state x to state y.
- **Solution:** A sequence of actions that leads from the initial state to a goal state.
- **Search Tree:** A tree data structure generated during the search process. The root is the initial state, and branches represent actions. A search tree is derived from the state space graph.
- **Node:** A data structure in the search tree, representing a state. A node contains:
  - **state:** The state in the state space to which the node corresponds.
  - **parent:** The node in the search tree that generated this node.
  - **action:** The action that was applied to the parent to generate the node.
  - **depth:** The number of steps along the path from the initial state.
  - **path_cost:** (Often denoted as $g(n)$) The cost of the path from the initial state to the node.
- **Fringe (or Frontier/Open List):** A data structure that keeps track of the nodes that have been generated but not yet expanded (i.e., their successors have not been explored).
- **Closed List (Explored Set):** Is to record every expanded node.

- **Expanding a Node:** Generating all the successor nodes of a given node by applying the successor function.
- **Search Strategy:** The method used to choose which node from the fringe to expand next. This is the key difference between different search algorithms.
- **Problem Types:**
  - **Single-state:** Deterministic and fully observable.
  - **Sensorless (Conformant):** Non-observable.
  - **Contingency:** Non-deterministic and/or partially observable.
  - **Exploration:** Unknown state space.

2 Uninformed Search Strategies

- Uninformed search strategies only use the information available in the problem definition (no extra heuristics).

2.1 Breadth-First Search (BFS):

Strategy: Expand the shallowest unexpanded node (FIFO queue for fringe).
Algorithm (simplified):

```
def BFS(problem):
    // Use a Queue (FIFO) for the fringe. This ensures shallowest nodes are expanded first.
        fringe = Queue()
        fringe.enqueue(Make-Node(Initial-State[problem]))
        closed = Set()  // Keep track of explored states
        while fringe is not empty:
            node = fringe.dequeue()  // Get the *oldest* node from the fringe
            if Goal-Test[problem](State[node]) then
                return Solution(node) // Found a solution!
            if State[node] is not in closed then
                add State[node] to closed
                for each successor in Expand(node, problem) do
                // Add *all* successors to the fringe, regardless of cost.
                    fringe.enqueue(successor)
    return failure  // No solution found
```

Properties:

- **Completeness:** Yes (if the branching factor b is finite).
- **Time Complexity:** $O(b^{d+1})$, where b is the branching factor and d is the depth of the shallowest goal.
- **Space Complexity:** $O(b^{d+1})$ (keeps all nodes in memory).
- **Optimality:** Yes, if all step costs are equal (otherwise, not optimal).

2.2 Uniform-Cost Search (UCS):

Strategy: Expand the node with the lowest path cost $g(n)$ (priority queue for fringe, ordered by $g(n)$).
Algorithm (simplified): Similar to BFS, but uses a priority queue ordered by path cost.

```
def uniform_cost_search(problem):
    """Uniform-Cost Search (UCS)."""
    # Use a priority queue: (cost, state, path)
    fringe = []  # List will be used as a heap
```

```
    initial_node = (0, problem.initial_state, [])  # (cost, state, path)
    heapq.heappush(fringe, initial_node)
    visited = set()

    while fringe:
        cost, current_state, path = heapq.heappop(fringe)

        if problem.is_goal_state(current_state):
            return path + [current_state]

        if current_state in visited:
            continue
        visited.add(current_state)

        for action, successor_state in problem.successor_function(current_state):
            if successor_state not in visited:
                new_path = path + [current_state]
                new_cost = cost + problem.cost(current_state, action, successor_state)
                heapq.heappush(fringe, (new_cost, successor_state, new_path))

    return failure
```

Properties:

- **Completeness:** Yes (if step costs are $\geq \epsilon > 0$, where ϵ is a small positive constant).
- **Time Complexity:** $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$, where C^* is the cost of the optimal solution. This can be much worse than BFS if step costs vary significantly.
- **Space Complexity:** $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$.
- **Optimality:** Yes.

2.3 Depth-First Search (DFS):

Strategy: Expand the deepest unexpanded node (LIFO stack for fringe).
Algorithm (simplified):

```
def DFS(problem):
    // Use a Stack (LIFO) for the fringe. This makes it expand the deepest nodes first.
    fringe = Stack()
    fringe.push(Make-Node(Initial-State[problem]))
    closed = Set()
    while fringe is not empty:
        node = fringe.pop()  // Get the *newest* node from the fringe
        if Goal-Test[problem](State[node]) then
            return Solution(node)
        if State[node] is not in closed then
            add State[node] to closed
            for each successor in Expand(node, problem) do
            // Add successors to the *top* of the stack.
                fringe.push(successor)
    return failure
```

Properties:

- **Completeness:** No (fails in infinite-depth spaces or with loops; can be made complete for finite spaces by checking for repeated states).
- **Time Complexity:** $O(b^m)$, where m is the maximum depth of the search tree (can be much larger than d).
- **Space Complexity:** $O(bm)$ (linear space).
- **Optimality:** No.

2.4 Depth-Limited Search (DLS):

Strategy: DFS with a predetermined depth limit l.
Algorithm: DFS, but nodes at depth l are not expanded.

```
def DLS(problem, limit):
        return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

def Recursive_DLS(node, problem, limit):
        if Goal-Test[problem](State[node]) then
            return Solution(node)  // Found a solution!
        else if Depth[node] = limit then
            return cutoff        // Reached the depth limit, no solution here
        else
            cutoff_occurred? ← false
            for each successor in Expand(node, problem) do
                result ← Recursive-DLS(successor, problem, limit)
                if result = cutoff then
                    cutoff_occurred? ← true   //Mark that we hit the limit somewhere
                else if result != failure then  // if it is solution
                    return result       // Found a solution in a deeper level!
            if cutoff_occurred? then
                return cutoff //Propagate cutoff if limit was reached
            else
                return failure // No solution found at this branch
```

```
def DLS_GraphSearch(problem, limit):
        fringe = Stack()
        fringe.push(Make-Node(Initial-State[problem]))
        closed = Set()
        while fringe is not empty:
            node = fringe.pop()
            if Goal-Test[problem](State[node]) then
                return Solution(node)
            if Depth[node] = limit then
                cutoff_occurred = True // Use this flag for track the depth
                continue //Skip the rest loop
            if State[node] is not in closed:
                add State[node] to closed
                cutoff_occurred = False
                for each successor in Expand(node, problem) do
                    result = fringe.push(successor)
                    if result = cutoff then
                        cutoff_occurred = True
                if cutoff_occurred then //This check the cutoff of all child
                    return cutoff
    return failure
```

Properties:

- **Completeness:** No (unless the solution is within the depth limit).
- **Time Complexity:** $O(b^l)$.
- **Space Complexity:** $O(bl)$.
- **Optimality:** No.

2.5 Iterative Deepening Search (IDS):

Strategy: Repeatedly runs DLS with increasing depth limits (0, 1, 2, ...).
Algorithm:

```python
def iterative_deepening_search(problem):
    """Iterative Deepening Search (IDS)."""
    for depth in range(0, float('inf')):  # In practice, you'd have a maximum depth
        result = depth_limited_search(problem, depth)
        if result != "cutoff":
            return result
```

Properties:

- **Completeness:** Yes (combines the benefits of BFS and DFS).
- **Time Complexity:** $O(b^d)$. Although it seems wasteful to repeat searches, the overhead is often small.
- **Space Complexity:** $O(bd)$ (linear space).
- **Optimality:** Yes (if step costs are equal).

3 Key Formulas and Notations

- b: Branching factor (maximum number of successors of any node).
- d: Depth of the shallowest goal node.
- m: Maximum depth of the state space (may be infinite).
- l: Depth limit (in DLS).
- C^*: Cost of the optimal solution.
- ϵ: Smallest positive step cost (used in UCS completeness analysis).
- $g(n)$: Path cost from the start node to node n.
- $\Gamma(n)$: Successor nodes of node n.
- N_{DLS}: Number of nodes generated by DLS. $N_{\text{DLS}} = b^0 + b^1 + \cdots + b^{d-1} + b^d$.
- N_{IDS}: Number of nodes generated by IDS. $N_{\text{IDS}} = (d+1)b^0 + db^1 + \cdots + 2b^{d-1} + b^d$.

4 Graph Search vs. Tree Search

- **Tree Search:** The basic search algorithms described above. They can repeatedly visit the same state if it's reachable via multiple paths.

```python
"""
Generic tree search algorithm.  The specific behavior depends on how 'fringe' is managed.
Args:
    problem:  The problem to be solved (defines initial state, goal test, successor function).
    fringe:   The data structure holding the nodes to be explored.
Returns:
    A solution (path from initial state to goal) or failure.
"""
def tree_search(problem, fringe):  # 'fringe' can be a list, queue, stack, or priority queue
    fringe.append(problem.initial_state)  # Add the initial state to the fringe
    while not fringe.is_empty():  # is_empty() is a placeholder for the appropriate method
        current_node = fringe.remove()  # remove() depends on fringe type (pop, dequeue, etc.)
        if problem.is_goal_state(current_node):
            return problem.solution(current_node)  # Return the path to the goal
        successors = problem.successor_function(current_node)  # Get successor states
        for successor in successors:
            fringe.append(successor)  # Add successors to the fringe
return failure  # No solution found
```

- **Graph Search:** An optimization that avoids revisiting states. It uses a closed set (or "explored" set) to keep track of nodes that have already been expanded. This prevents redundant work and guarantees completeness even in state spaces with loops.

```python
"""
Graph search algorithm, which avoids revisiting states.  This can be used
with different fringe types to implement BFS, DFS, or UCS (with appropriate
fringe management).

The main difference with Tree Search is the "closed" set
"""
def graph_search(problem, fringe):
    closed = set()  # Keep track of visited states
    fringe.append((problem.initial_state, [])) # (state, path)
    while fringe:
        if isinstance(fringe, list):
          current_state, path = fringe.pop(0) # Use pop(0) for queue-like, pop() for stack
        else:
          current_state, path = fringe.pop()
        if problem.is_goal_state(current_state):
            return path + [current_state]
        if current_state in closed:
            continue #skip if visited
        closed.add(current_state)
        for action, next_state in problem.successor_function(current_state):
            new_path = path + [current_state] # build the path
            fringe.append((next_state, new_path))
    return failure
```

5 Summary Table

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

Table 1: Comparison of Search Algorithms

Chapter 4: Informed Searching

1 Informed Search (Heuristic Search)

Key Idea: Use problem-specific knowledge (heuristics) to guide the search, making it more efficient than uninformed search.

1.1 Evaluation Function ($f(n)$)

A function that estimates the "desirability" of a node n. This is the core of informed search. The search algorithm prioritizes nodes with better $f(n)$ values.

1.2 Heuristic Function ($h(n)$)

Estimates the cost of the cheapest path from node n to a goal node. This is the "problem-specific knowledge."

1.3 Best-First Search

A general class of search algorithms that use an evaluation function $f(n)$ to select the next node to expand. Nodes are typically kept in a priority queue ordered by $f(n)$.
Implementation: Order nodes in the fringe in descending order of "suitability" (usually ascending order of estimated cost).

1.4 Greedy Best-First Search

Evaluation Function: $f(n) = h(n)$ (uses only the heuristic).
Strategy: Expands the node that appears to be closest to the goal (based on the heuristic).
Algorithm:

```python
"""
Greedy Best-First Search algorithm.
Args:
    problem: Defines initial_state, is_goal_state, successor_function, solution, and h(n).
Returns:
    A solution or failure.
"""
def greedy_best_first_search(problem):
    # (heuristic, state, path)
    initial_node = (problem.h(problem.initial_state), problem.initial_state, [])
    fringe = [initial_node]  # Priority queue: (priority, item)
    heapq.heapify(fringe) #make fringe to become the heap
    visited = set()       # Keep track of visited states (for efficiency)
    while fringe:
        _, current_state, path = heapq.heappop(fringe)  # Get node with lowest heuristic
        if problem.is_goal_state(current_state):
            return path + [current_state]
        if current_state in visited:
            continue
        visited.add(current_state)
        for successor in problem.successor_function(current_state):
            if successor not in visited:
                new_path = path + [current_state]
                priority = problem.h(successor)   # f(n) = h(n) for Greedy Best-First
                heapq.heappush(fringe, (priority, successor, new_path))
    return failure
```

Properties:

- *Not Complete:* Can get stuck in loops.
- *Not Optimal:* Doesn't consider the cost to reach the node.
- *Time and Space Complexity:* $O(b^m)$ in the worst case, but a good heuristic can significantly improve this.

1.5 A* Search

Evaluation Function: $f(n) = g(n) + h(n)$

- $g(n)$: Cost of the path from the start node to node n.
- $h(n)$: Heuristic estimate of the cost from node n to the goal.
- $f(n)$: Estimated cost of the cheapest solution through node n.

Strategy: Expands the node with the lowest estimated total cost ($f(n)$). Combines the cost-so-far ($g(n)$) with the estimated cost-to-go ($h(n)$).
Algorithm:

```python
"""
A* Search algorithm.
Args:
    problem: Defines initial_state, is_goal_state, successor_function, solution, h(n), and g(n).
Returns:
    A solution or failure.
"""
def a_star_search(problem):
    # (f(n), g(n), state, path)
    initial_node = (problem.h(problem.initial_state), 0, problem.initial_state, [])
    fringe = [initial_node]
    heapq.heapify(fringe) #make fringe to become the heap
    visited = set()
    while fringe:
        _, g, current_state, path = heapq.heappop(fringe) #get node with lowest f
        if problem.is_goal_state(current_state):
            return path + [current_state]
        if current_state in visited:
            continue
        visited.add(current_state)
        for successor in problem.successor_function(current_state):
            if successor not in visited:
                new_path = path + [current_state]
                new_g = g + problem.cost(current_state, successor) # Calculate g(n) for successor
                priority = new_g + problem.h(successor)          # f(n) = g(n) + h(n)
                heapq.heappush(fringe, (priority, new_g, successor, new_path))
    return failure
```

Properties:

- If the state space is finite and there is a solution, then A* is complete, but not optimal.
- Otherwise, A* is incomplete.

1.5.1 Admissible Heuristic

Definition: A heuristic $h(n)$ is admissible if it never overestimates the true cost to reach the goal. Formally: $0 \leq h(n) \leq h^*(n)$ for all nodes n, where $h^*(n)$ is the true cost.
Importance: If $h(n)$ is admissible, A* search using tree search is optimal.

1.5.2 Consistent Heuristic (Monotonic Heuristic)

Definition: A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by action a: $h(n) \leq c(n, a, n') + h(n')$. This is a stronger condition than admissibility. It's a form of the triangle inequality.
Importance: If $h(n)$ is consistent, A* search using graph search is optimal. Furthermore, $f(n)$ values along any path are non-decreasing.
If h is consistent, then:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

1.5.3 Dominant Heuristic

If $h_2(n) \geq h_1(n)$ for all n (and both are admissible), h_2 dominates h_1 and is better for search.

2 Local Search

Key Idea: Instead of systematically exploring the state space, local search algorithms start with an initial state and iteratively try to improve it by making "local" changes. They don't keep track of the path, only the current state.

When to Use: Useful for problems where the path to the goal is irrelevant, and only the final state matters (e.g., optimization problems, constraint satisfaction problems).
State Space: A set of "complete" configurations.
Goal: Find a configuration that satisfies constraints or optimizes an objective function.

2.1 Hill-Climbing Search (Gradient Ascent/Descent)

Strategy: Always move to a neighbor with a better value of the objective function. If no such neighbor exists, the algorithm terminates (at a local optimum).
Algorithm (Simplified):

```
"""
Hill-climbing search algorithm.
Args:
    problem: Defines initial_state, neighbor function (or successor_function),
             and value function (objective function to maximize).
Returns:
    The "best" state found (a local maximum).
"""
def hill_climbing_search(problem):
    current_state = problem.initial_state
    while True:
        neighbors = problem.successor_function(current_state)
        # Or problem.neighbors(current_state)
        if not neighbors:  # No neighbors (can happen in discrete spaces)
            return current_state
        best_neighbor = None
        best_neighbor_value = float('-inf')
        for neighbor in neighbors:
            neighbor_value = problem.value(neighbor)
            if neighbor_value > best_neighbor_value:
                best_neighbor = neighbor
                best_neighbor_value = neighbor_value
        if best_neighbor_value <= problem.value(current_state):
            # No better neighbor found; we're at a local maximum (or plateau)
            return current_state
        current_state = best_neighbor
```

Issues: Can get stuck in local optima (maxima or minima), shoulders, and plateaus.

2.2 Simulated Annealing

Key Idea: Allow "bad" moves (moves that worsen the objective function) with a probability that decreases over time. This helps escape local optima.
Analogy: Inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to reach a low-energy, stable state.
Temperature (T): A control parameter that determines the probability of accepting bad moves. High T means more bad moves are accepted; low T means fewer are accepted.
Algorithm (Simplified):

```
"""
Simulated annealing algorithm.
Args:
    problem:  Defines the state space, neighbor function, and value function.
    schedule: A function that maps time (iteration number) to temperature.
Returns:
    The best state found.
"""
def simulated_annealing(problem, schedule):
    current_state = problem.initial_state
    t = 1  # Iteration counter
```

```
    while True:  # Termination condition could also be based on a fixed number of iterations
        T = schedule(t)  # Get the current temperature
        if T == 0:
            return current_state
        next_state = problem.random_neighbor(current_state)  # Get a random neighbor
        delta_e = problem.value(next_state) - problem.value(current_state)  # Change in value
        if delta_e > 0:  # Improvement
            current_state = next_state
        else:
            probability = math.exp(delta_e / T)  # Probability of accepting a worse state
            if random.random() < probability:  # random.random() returns a float in [0.0, 1.0)
                current_state = next_state
        t += 1
```

Properties: If T decreases slowly enough, simulated annealing can find the global optimum with probability approaching 1.

2.3 Local Beam Search

Key Idea: Keep track of k states (instead of just one) at each iteration. Generate all successors of all k states, and then select the k best successors to continue.
Difference from k random restarts: In local beam search, the k states "communicate" with each other by generating all their successors and selecting the best overall.
Algorithm:

```
"""
Local beam search algorithm.
Args:
    problem: Defines the state space, successor function, and value function.
    k: The number of states to keep track of.
Returns:
    The best state found.
"""
def local_beam_search(problem, k):
    states = [problem.random_state() for _ in range(k)]  # k random initial states
    while True:
        # Could also terminate based on a fixed number of iterations or lack of improvement
        all_successors = []
        for state in states:
            successors = problem.successor_function(state)
            all_successors.extend(successors)
        if not all_successors: # No successors, stop.  (Avoids errors later)
            return max(states, key=problem.value)
        #check goal
        for successor in all_successors:
            if problem.is_goal_state(successor):
                return successor
        # Select the k best successors
        states = sorted(all_successors, key=problem.value, reverse=True)[:k]
        # Check for stagnation (optional, but often used)
        if all(problem.value(s) <= problem.value(states[0]) for s in all_successors):
            return max(states,key=problem.value)
```

3 Adversarial Search (Games)

Key Idea: Deals with search problems in environments with multiple agents whose goals may conflict. Often used in games.
Zero-Sum Games: A game where the total payoff to all players is constant for all outcomes. One player's gain is another player's loss.

3.1 Game Tree

- **Nodes:** Represent game states.
- **Edges:** Represent moves.
- **Levels:** Alternate between players (e.g., MAX and MIN).
- **Initial State:** The starting state of the game, including who moves first.
- **Successor Function:** Returns a list of (move, state) pairs, indicating the legal moves and resulting states.
- **Terminal Test:** Determines if the game is over.
- **Utility Function (Payoff Function):** Assigns a numerical value to terminal states (e.g., +1 for a win, −1 for a loss, 0 for a draw).

3.2 MINIMAX Algorithm

Goal: Find the optimal strategy for MAX, assuming MIN also plays optimally.

MINIMAX Value

Concept: A recursive value assigned to each node in a game tree, assuming both players play optimally.

- **MAX:** Chooses moves to maximize the MINIMAX value.
- **MIN:** Chooses moves to minimize the MINIMAX value.

Recursive Definition:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } s \text{ is a terminal state} \\ \max(\text{MINIMAX}(s')) & \text{for all successors } s' \text{ of } s, \text{ if } s \text{ is a MAX node} \\ \min(\text{MINIMAX}(s')) & \text{for all successors } s' \text{ of } s, \text{ if } s \text{ is a MIN node} \end{cases}$$

Idea: Performs a depth-first search of the game tree to compute the MINIMAX value of each node, ultimately determining the best move for the current player.
Algorithm:

```
"""
Minimax algorithm for adversarial search.
Args:
    node: The current game state.
    depth: The remaining search depth.
    maximizing_player: True if it's MAX's turn, False if it's MIN's turn.
    problem:  Defines game rules (successor function, utility function, terminal test).
Returns:
    The minimax value of the node.
"""
def minimax(node, depth, maximizing_player, problem):
    if depth == 0 or problem.is_terminal(node):
        return problem.utility(node, maximizing_player)  # Utility for the maximizing player
    if maximizing_player:
        best_value = float('-inf')
        for child in problem.successor_function(node):
            value = minimax(child, depth - 1, False, problem)
            best_value = max(best_value, value)
        return best_value
    else:
        best_value = float('inf')
        for child in problem.successor_function(node):
            value = minimax(child, depth - 1, True, problem)
            best_value = min(best_value, value)
```

```
        return best_value
def get_best_move_minimax(problem, current_state, max_depth):
    """Helper function to determine the best move using Minimax."""
    best_move = None
    best_value = float('-inf')
    for move, new_state in problem.successor_function(current_state):
        value = minimax(new_state, max_depth, False, problem)  # False, as next player is MIN
        if value > best_value:
            best_value = value
            best_move = move
    return best_move
```

Properties:

- *Complete:* Yes (if the tree is finite).
- *Optimal:* Yes (against an optimal opponent).
- *Time Complexity:* $O(b^m)$.
- *Space Complexity:* $O(bm)$ (for depth-first exploration).

3.3 Alpha-Beta Pruning

Key Idea: Avoid exploring parts of the game tree that cannot possibly influence the final decision. This dramatically improves efficiency.
Alpha (α): The best value (highest) found so far for MAX along the path.
Beta (β): The best value (lowest) found so far for MIN along the path.
Pruning Rule:

- If a MAX node has a value v such that $v \geq \beta$, then the branch below that MAX node can be pruned (Cut off a branch below a MAX node, because MIN will never choose to go to that branch).
- If a MIN node has a value v such that $v \leq \alpha$, then the branch below that MIN node can be pruned (Cut off a branch below a MIN node, because MAX will never choose to go to that branch).

Effectiveness: Can significantly reduce the search space, but the effectiveness depends on the order in which nodes are explored. Best case: $O(b^{m/2})$ time complexity.
Algorithm:

```
"""
Alpha-beta pruning algorithm (optimization of Minimax).
Args:
    node: The current game state.
    depth: The remaining search depth.
    alpha: The best value found so far for MAX along the path.
    beta:  The best value found so far for MIN along the path.
    maximizing_player: True if it's MAX's turn, False if it's MIN's turn.
    problem:  Defines game rules (successor function, utility, terminal test).
Returns:
    The minimax value of the node, considering alpha-beta pruning.
"""
def alpha_beta(node, depth, alpha, beta, maximizing_player, problem):
    if depth == 0 or problem.is_terminal(node):
        return problem.utility(node, maximizing_player)
    if maximizing_player:
        value = float('-inf')
        for child in problem.successor_function(node):
            value = max(value, alpha_beta(child, depth - 1, alpha, beta, False, problem))
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # Beta cutoff
        return value
```

```
    else:
        value = float('inf')
        for child in problem.successor_function(node):
            value = min(value, alpha_beta(child, depth - 1, alpha, beta, True, problem))
            beta = min(beta, value)
            if beta <= alpha:
                break  # Alpha cutoff
        return value
def get_best_move_alpha_beta(problem, current_state, max_depth):
    """Helper function to determine best move using Alpha-Beta pruning."""
    best_move = None
    best_value = float('-inf')
    for move, new_state in problem.successor_function(current_state):
        # next player is MIN
        value = alpha_beta(new_state, max_depth, float('-inf'), float('inf'), False, problem)
        if value > best_value:
            best_value = value
            best_move = move
        return best_move
```

4 Key Formulas

- **Greedy Best-First Search:** $f(n) = h(n)$
- **A* Search:** $f(n) = g(n) + h(n)$
- **Admissible Heuristic:** $0 \leq h(n) \leq h^*(n)$
- **Consistent Heuristic:** $h(n) \leq c(n, a, n') + h(n')$
- **MINIMAX Value (Recursive):** See above.
- **Simulated Annealing:** Probability $\exp\left(\frac{\Delta E}{T}\right)$

Chapter 5: Constraint Satisfaction

1 Core Theory & Definitions

- **Constraint:** A relation or restriction on a set of variables (e.g., $X < Y$, Sum of angles $= 180°$, Adjacent regions $\neq$ same color).
- **Variable:** A placeholder that needs to be assigned a value.
- **Domain:** The set of possible values a variable can take (e.g., {red, green, blue}, {0, 1, ..., 9}, {true, false}). These slides focus on finite, discrete domains.
- **Constraint Satisfaction Problem (CSP):** Defined by three components:
  - X: A finite set of Variables.
  - D: A set of Domains, one for each variable.
  - C: A finite set of Constraints restricting the values variables can take simultaneously.
- **Solution to a CSP:** A complete (every variable assigned) and consistent (all constraints satisfied) assignment of values to variables.
- **Constraint Graph:** A visual representation (often for binary CSPs) where nodes are variables and arcs (edges) represent constraints between them.

2 Types of CSPs & Constraints

2.1 Based on Variables/Domains:

- **Discrete Variables with Finite Domains:** The most common type studied (e.g., Map Coloring). Complexity is $O(d^n)$ for n variables and domain size d.
- **Discrete Variables with Infinite Domains:** (e.g., integers, strings for scheduling). Requires a constraint language.
- **Continuous Variables:** Domains are real numbers (e.g., physics simulations, some scheduling). Requires different techniques (often numerical).

2.2 Based on Constraint Arity (Number of involved variables):

- **Unary:** Involves a single variable (e.g., SA $\neq$ green).
- **Binary:** Involves pairs of variables (e.g., SA $\neq$ WA). Most common, allow constraint graph representation.
- **Higher-order:** Involves 3 or more variables (e.g., cryptarithmetic constraints like $O + O = R + 10 \cdot X_1$).

3 Key Problem-Solving Approaches & Techniques for CSPs

3.1 Generate and Test (Basic but Inefficient):

- Generate a complete assignment candidate.
- Test if it violates any constraints.
- **Weakness:** Tries many obviously invalid assignments, doesn't learn from failures early.

3.2 Backtracking Search (Standard Systematic Approach):

- Improvement over Generate and Test.
- Based on Depth-First Search (DFS).
- Assigns values to one variable at a time.
- Checks consistency only with already assigned variables after each assignment.
- If an assignment violates a constraint, backtrack: undo the assignment and try the next value.
- **Algorithm:** `RECURSIVE-BACKTRACKING(assignment, csp)` function (Slide 17).
- **Weakness:** Can still detect contradictions late (only when an assignment directly causes a violation).

3.3 Improving Backtracking Efficiency (Heuristics & Propagation):

- **Variable Ordering: Which variable to assign next?**
  - **Minimum Remaining Values (MRV) / Most Constrained Variable:** Choose the variable with the fewest legal values left in its domain. (Slide 25)
  - **Degree Heuristic (Tie-breaker for MRV):** Choose the variable involved in the most constraints with other unassigned variables. (Slide 26)
- **Value Ordering: In what order to try values for the selected variable?**
  - **Least Constraining Value (LCV):** Prefer the value that rules out the fewest choices for neighboring (constrained) variables in the future. (Slide 27)
- **Early Detection of Contradictions (Constraint Propagation):**

- **Forward Checking:** When assigning Var = value, check constraints involving Var and its unassigned neighbors. Remove inconsistent values from neighbors' domains. Backtrack if any neighbor's domain becomes empty. (Slides 28-32). More proactive than basic backtracking but doesn't detect all inconsistencies.
- **Arc Consistency (AC-3 Algorithm):** Stronger form of propagation. An arc $X \to Y$ is arc-consistent if for every value x in Domain(X), there is some allowed value y in Domain(Y). Enforcing arc consistency involves removing values from domains that cannot satisfy constraints. This propagates. (Slides 34-38). Can be done before search or interleaved (Maintaining Arc Consistency - MAC).

3.4 Local Search (Alternative Approach):

- Operates on complete assignments, even inconsistent ones.
- Starts with a random complete assignment.
- Iteratively modifies the value of one variable at a time to reduce conflicts.
- **State:** A complete assignment.
- **Evaluation Function** (e.g., for Hill Climbing): $h(n)$ = number of violated constraints.
- **Key Heuristic: Min-Conflicts:** When choosing a new value for a variable, select the value that results in the minimum number of conflicts with other variables. (Slide 40)
- Often very effective for large CSPs, but may get stuck in local optima (not guaranteed to find a solution if one exists). Examples: Hill-climbing, Simulated Annealing.

4 Fundamental Concepts & Keys

- **Problem Formulation:** Recognizing that a problem can be framed as a CSP is crucial. Identify variables, domains, and constraints.
- **Trade-offs:** Systematic search (Backtracking) is complete (guaranteed to find a solution if one exists) but can be slow. Local search is often faster and scales better but is incomplete.
- **Heuristics:** Intelligent choices (variable/value ordering, min-conflicts) are critical for making search feasible in practice.
- **Constraint Propagation:** Looking ahead (Forward Checking, Arc Consistency) prunes the search space significantly by eliminating impossible assignments early.

Chapter 6: Logic and Reasoning

1 Core Concepts of Logic

1.1 What is Logic?

- Formal languages for representing information precisely.
- Allows drawing valid conclusions (inferences) from known information.
- Formula: Logic = Syntax + Semantics

1.2 Syntax:

- Defines the rules for constructing well-formed sentences (formulas) in the language. It's about structure, not meaning.
- Includes:
  - **Language:** Legal symbols, how to combine them into terms, expressions, formulas (e.g., $(x + 2 \geq y)$ is syntactically correct, $(x + y > \{\})$ is not).

  - **Proof Theory:** A set of inference rules that allow deriving new sentences (theorems) from existing ones based purely on their syntactic form.

- Example Inference Rule: any plus zero $\vdash$ any (The symbol $\vdash$ denotes syntactic derivation/proof).
- Proving theorems syntactically doesn't require knowing the meaning (interpretation) of the symbols.

1.3 Semantics:

- Defines the meaning or truth of sentences.
- Connects the symbols and formulas to the actual world or a conceptual model.
- Requires an Interpretation (I) which assigns meaning:
  - I(symbol) maps symbols to objects, relations, or functions in a domain (e.g., I(one) means the number 1, I(plus) means the addition operation +).
- Semantics defines how to determine the truth value (true/false) of a sentence within a given interpretation.

2 Key Logical Relationships

2.1 Model:

- An interpretation m is a model of a sentence α if sentence α is true in that interpretation m.
- $M(\alpha)$ represents the set of all models for sentence α.

2.2 Entailment ($\models$):

- The most fundamental concept: What conclusions logically follow from what we know?
- Definition: A knowledge base KB entails a sentence α (KB $\models \alpha$) if and only if α is true in every interpretation (world) where all sentences in KB are true.
- In other words: If KB is true, α must also be true.
- Based purely on semantics (meaning/truth).
- Model-theoretic view: KB $\models \alpha$ if and only if $M(\text{KB}) \subseteq M(\alpha)$ (The set of models of KB is a subset of the set of models of α).

3 Logical Inference (Derivation)

3.1 Inference ($\vdash$):

- The process of deriving new sentences from existing ones using inference rules (part of syntax/proof theory).
- KB $\vdash_i \alpha$: Sentence α can be derived (inferred) from KB using inference procedure i.

3.2 Soundness:

- An inference procedure i is sound if it only derives sentences that are actually entailed.
- Formally: If KB $\vdash_i \alpha$, then KB $\models \alpha$.
- Crucial: Unsound procedures derive falsehoods!

3.3 Completeness:

- An inference procedure i is complete if it can derive every sentence that is entailed.
- Formally: If KB $\models \alpha$, then KB $\vdash_i \alpha$.
- Desirable: Ensures we can eventually prove anything that logically follows.

3.4 Inference Methods:

- **Deductive Reasoning (Proofs):** Manipulating sentences syntactically using inference rules.
- **Model-Based Reasoning:** Working directly with interpretations/models (semantics).
  - Model Checking: Given α and interpretation I, is I a model of α?
  - Satisfiability: Given α, does there exist any model for α?
  - Validity Checking: Given α, is α true in all possible interpretations?

4 Propositional Logic (PL)

4.1 Basics:

- The simplest logic. Deals with propositions (sentences that are either true or false).
- Symbols: P, Q, R... representing propositions.
- Constants: true, false.

4.2 PL Syntax:

- Atomic sentences: Symbols (P) or constants (true).
- Complex sentences built using logical connectives:
  - $\neg S$ (Negation: "not")
  - $S_1 \wedge S_2$ (Conjunction: "and")
  - $S_1 \vee S_2$ (Disjunction: "or")
  - $S_1 \Rightarrow S_2$ (Implication: "implies", "if S_1 then S_2")
  - $S_1 \Leftrightarrow S_2$ (Equivalence: "if and only if")
- Operator Precedence: $\neg$ (highest), $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ (lowest). Parentheses override.

4.3 PL Semantics:

- An interpretation assigns a truth value (true or false) to each propositional symbol.
- Truth Tables define the semantics of the connectives for any given interpretation (Slide 20). Memorize these rules:
  - $\neg S_1$: True iff S_1 is false.
  - $S_1 \wedge S_2$: True iff both S_1 and S_2 are true.
  - $S_1 \vee S_2$: True iff at least one of S_1 or S_2 is true.
  - $S_1 \Rightarrow S_2$: False iff S_1 is true and S_2 is false. (True otherwise).
  - $S_1 \Leftrightarrow S_2$: True iff S_1 and S_2 have the same truth value.

4.4 Key Concepts in PL:

- **Logical Equivalence ($\equiv$):** Two sentences are equivalent if they are true in the exact same models (interpretations). Slide 21 lists important equivalences (De Morgan's Laws, Distributivity, etc.). $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$.
- **Contradiction:** A sentence false in every interpretation (e.g., $P \wedge \neg P$).
- **Tautology (Valid Sentence):** A sentence true in every interpretation (e.g., $P \vee \neg P$).
- **Satisfiable:** A sentence true in at least one interpretation.
- **Unsatisfiable:** A sentence true in no interpretation (a contradiction).

5 Proving Entailment in Propositional Logic

5.1 The Logic Proving Problem:

- Given KB, does KB $\models \alpha$?

5.2 Proving Methods:

5.2.1 Truth Table Enumeration: (Slides 27-29)

- List all possible interpretations (2^n rows for n symbols).
- Check if α is true in every row where all sentences in KB are true.
- Pros: Sound and Complete for PL.
- Cons: Computationally intractable ($O(2^n)$).

5.2.2 Inference Rules (Syntactic Deduction):

- Apply a sequence of sound inference rules starting from KB to derive α.
- Common Rules:
  - Modus Ponens: $(A \Rightarrow B), A \vdash B$
  - And-Elimination: $A \wedge B \vdash A$ (and $\vdash B$)
  - And-Introduction: $A, B \vdash A \wedge B$
  - Or-Introduction: $A \vdash A \vee B$
  - Double Negation: $\neg\neg A \vdash A$
  - Unit Resolution: $A \vee B, \neg B \vdash A$
  - Resolution: $A \vee B, \neg B \vee C \vdash A \vee C$ (Key rule for resolution proving)
- Finding the right sequence is a search problem.

5.2.3 Resolution Refutation (via SAT):

- The most common practical approach for automated theorem proving in PL.
- Key Idea: KB $\models \alpha$ if and only if (KB $\wedge\neg\alpha$) is unsatisfiable (a contradiction).
- Steps (Robinson's Algorithm):
  1. Convert all sentences in KB and the negation of the goal ($\neg\alpha$) into Conjunctive Normal Form (CNF).
     - CNF: A conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals (e.g., $(P \vee \neg Q) \wedge (\neg R \vee S)$).
     - Conversion Steps (Slide 37): Eliminate $\Leftrightarrow$, $\Rightarrow$; Move $\neg$ inwards (De Morgan); Distribute $\vee$ over $\wedge$.

2. Combine all CNF clauses into a single set.
3. Repeatedly apply the Resolution Rule ($A \vee B, \neg B \vee C \vdash A \vee C$) to pairs of clauses containing complementary literals (e.g., B and $\neg B$). Add the resolvent (result) to the set.
4. If the empty clause ($\{\}$ or $\perp$) is derived, it signifies a contradiction. This means (KB $\wedge \neg\alpha$) is unsatisfiable, therefore KB $\models \alpha$ is proven.
5. If no new clauses can be derived and the empty clause wasn't found, then KB $\models \alpha$ is not proven.

- **SAT Problem (Satisfiability):** Determining if a CNF formula is satisfiable. SAT solvers are highly optimized algorithms for this, often using backtracking (like DPLL) or local search (like WalkSAT). (Slides 39-40). Resolution refutation is essentially one way to check for unsatisfiability.

6 Horn Clauses and Chaining

6.1 Horn Clauses:

- A special type of clause: A disjunction of literals with at most one positive literal.
- Forms:
  - **Definite Clause (Rule):** $(P_1 \wedge ... \wedge P_k) \Rightarrow Q$ (Exactly one positive literal Q)
  - **Fact:** P (Exactly one positive literal, no negative ones)
  - **Integrity Constraint:** $(P_1 \wedge ... \wedge P_k) \Rightarrow$ false (No positive literals)
- Important because inference with Horn clauses is computationally simpler (can be done in linear time).

6.2 Generalized Modus Ponens (GMP):

- Rule: $(P_1 \wedge ... \wedge P_k \Rightarrow Q), P_1, ..., P_k \vdash Q$
- Sound and complete for Knowledge Bases containing only Horn Clauses.

6.3 Reasoning with Horn Clauses:

6.3.1 Forward Chaining (Data-Driven):

- Start with known facts.
- Apply rules (GMP) whose premises are satisfied by current facts to add new facts.
- Repeat until the goal is derived or no new facts can be added.
- Good when new facts might trigger many conclusions (e.g., sensor readings). Can be unfocused.

6.3.2 Backward Chaining (Goal-Driven):

- Start with the goal query Q.
- Find rules that conclude Q. Recursively try to prove the premises of those rules (treat premises as sub-goals).
- Uses known facts to terminate recursion.
- More focused for specific queries (e.g., diagnosis, planning).

7 Limitations of Propositional Logic & Intro to First-Order Logic (FOL)

7.1 PL Limitations:

- Limited Expressiveness: Cannot represent objects, their properties, or relations between them abstractly. Cannot easily express generalizations like "All students..." or "Some people...". Requires creating specific propositions for every instance (e.g., Tuan_is_student, Cuong_is_student).

7.2 First-Order Logic (FOL):

- More expressive logic that overcomes PL's limitations.
- Introduces:
  - **Objects:** Things in the world (Tuan, Table, Red).
  - **Properties:** Unary relations (Student, Red).
  - **Relations:** N-ary relations (SisterOf, GreaterThan).
  - **Functions:** (FatherOf, SquareRoot).
  - **Variables:** x, y, z that range over objects.
  - **Quantifiers:** $\forall$ (Universal: "For all"), $\exists$ (Existential: "There exists").

8 First-Order Logic (FOL) Details

8.1 FOL Syntax:

- **Symbols:** Constants (Tuan), Variables (x), Function Symbols (plus), Predicate Symbols (Student, $>$).
- **Terms:** Refer to objects. A constant, a variable, or a function applied to terms (e.g., Tuan, x, plus(x, 2)).
- **Atoms:** Smallest unit with a truth value. A predicate applied to terms (e.g., Student(Tuan), GreaterThan(plus(x,2), 5)).
- **Formulas:** Atoms, or complex formulas built with logical connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) and quantifiers ($\forall x$: Formula, $\exists x$: Formula).

8.2 FOL Semantics:

- Requires an Interpretation $\langle D, I \rangle$:
  - D: A non-empty Domain of objects.
  - I: An interpretation function mapping:
    - Constants to objects in D.
    - Function symbols to actual functions $D^n \to D$.
    - Predicate symbols to actual relations $D^n \to \{\text{true, false}\}$.
- Truth of formulas evaluated relative to $\langle D, I \rangle$.

8.3 Quantifier Semantics:

- $\forall x : P(x)$ is true iff $P(d)$ is true for every object d in D.
- $\exists x : P(x)$ is true iff $P(d)$ is true for at least one object d in D.
- Satisfiable/Unsatisfiable/Valid: Defined similarly to PL, but over FOL interpretations.
- Model: An interpretation $\langle D, I \rangle$ that makes a formula true.

8.4 Using FOL:

- Allows representing complex knowledge much more naturally and compactly than PL (Slide 77 examples).

Chapter 7: Knowledge Representation

1 Data, Information, Knowledge Hierachy (DIKW Pyramid)

- **Data:** Raw, unorganized facts, symbols, or signals (e.g., `"5 degrees Celsius"`). Low value, often high volume, lacks context.
- **Information:** Data processed into a meaningful form, given context (e.g., `"It is cold outside"`). Higher value, lower volume than data.
- **Knowledge:** Understanding derived from information, internalizing relationships, patterns, and implications. Facilitates decisions and actions (e.g., `"If it is cold outside, then you should wear a warm coat"`). Highest value, integrates information.
- **Meta-Knowledge:** Knowledge *about* knowledge (e.g., knowing *when* or *how* to apply a piece of knowledge).
- **Key Idea:** Value increases as raw data is processed into information and then into actionable knowledge. KR focuses on representing the "Knowledge" and "Meta-Knowledge" levels.

2 Knowledge Representation (KR) Fundamentals

- **Goal:** Develop formal methods and tools to represent knowledge so it can be stored, retrieved, and reasoned with by AI systems.
- **Key KR Methods Covered:**
  - Production Rules
  - Frames
  - Semantic Networks
  - Ontology (also mentions probabilistic models, deep models)
- **Desirable Properties of KR Systems:**
  - **Completeness:** Can it represent all necessary knowledge in the domain?
  - **Conciseness:** Is the representation efficient to store and access?
  - **Computational Efficiency:** Can reasoning be performed efficiently?
  - **Transparency:** Is the representation and reasoning process understandable (to humans/experts)?

3 Production Rules (Rule-Based Representation)

- **Structure:** `IF <Conditions> THEN <Conclusion/Action>`
  - `Conditions (`A_i`)`: Antecedents or premises (conjunction: A_1 `AND` A_2`...AND` A_n). Matched against facts in Working Memory.
  - `Conclusion (B)`: Consequence or action. Added to Working Memory when the rule fires.
- **Simplifications:**
  - OR in conditions: `IF` A_1 `OR` A_2 `THEN B` becomes `IF` A_1 `THEN B` AND `IF` A_2 `THEN B`.
  - AND in conclusions: `IF...THEN` B_1 `AND` B_2 becomes `IF...THEN` B_1 AND `IF...THEN` B_2.
  - OR in conclusions is generally *not* allowed directly.
- **Types:** Represent associative, causal, situation-action, logical relations.
- **Representation:** Can be visualized using AND/OR graphs.
- **Usage:**

  - **Pattern Matching:** Determining which rules' conditions are met by current facts.
  - **Chaining:** Linking rules together to reach conclusions. Uses **Forward Chaining** (data-driven, from facts to goals) and **Backward Chaining** (goal-driven, from goals to facts) – covered in the Logic section.
  - **Conflict Resolution:** Strategy needed when multiple rules match the facts simultaneously. Strategies include: first applicable, most specific, most recent facts, highest certainty, avoid duplicates, combinations.
- **Rule-Based System Architecture:**
  - **Working Memory:** Holds current facts/data.
  - **Rule Memory (Knowledge Base):** Stores the production rules.
  - **Interpreter:** Matches rules, performs conflict resolution, fires rules (executes actions/adds conclusions), modifies working memory. (Match-Resolve-Act cycle).
- **Advantages:** Notational convenience (natural language like), easy to understand, declarative (separates knowledge from control).
- **Disadvantages:** Restricted expression power, difficult to manage rule interactions (hard to maintain/update), potentially expensive to build.

4 Frames (Frame-Based Representation)

- **Motivation:** Representing objects and their properties in a structured way (Object-Property-Value).
- **Structure:** `Frame` = Named structure representing an object or concept.
  - `Slots`: Represent properties or attributes of the frame.
  - `Fillers`: Values associated with slots.
- **Types:**
  - **Generic Frames:** Represent classes or categories (e.g., `CanadianCity`). Use `IS-A` links to more general frames.
  - **Individual Frames:** Represent specific instances (e.g., `toronto`). Use `INSTANCE-OF` links to generic frames.
- **Key Concepts & Inference:**
  - **Inheritance:** Individual frames inherit slots and fillers from their generic frames via `INSTANCE-OF` and `IS-A` links. More specific frames inherit from more general ones.
  - **Defaults:** Fillers in generic frames provide default values for instances, which can be overridden.
  - **Procedural Attachment (Demons):** Procedures attached to slots:
    - `IF-NEEDED`: Executed when a slot's value is required but not present (e.g., compute age from birthdate).
    - `IF-ADDED`: Executed when a value is added to a slot (e.g., update related information, check constraints).
- **Reasoning Process:** Instantiate frame -¿ Inherit slots/fillers -¿ Run inherited `IF-ADDED` procedures -¿ Query slot (use filler if present, else run inherited `IF-NEEDED` procedure).
- **Advantages:** Combines declarative (slots/fillers) and procedural knowledge (`IF-NEEDED`/`IF-ADDED`), hierarchical structure supports classification and defaults, reduces complexity.
- **Disadvantages:** Requires careful taxonomy design, can lead to complex procedural interactions ("procedural fever"), potentially inefficient runtime storage.

5 Semantic Networks (SN)

- **Structure:** Graph-based representation.
  - `Nodes`: Represent concepts, objects, actions.
  - `Links (Arcs)`: Labeled, directed edges representing relationships between nodes.
- **Key Link Types:**
  - `IS-A`: Subclass relationship (inheritance).
  - `INSTANCE-OF`: Instance relationship (inheritance).
  - Domain-Specific: `HAS-PART`, `COLOR`, `EATS`, `CAUSES`, etc.
- **Reasoning:**
  - **Inheritance:** Properties propagate down `IS-A`/`INSTANCE-OF` links (e.g., an Elephant `IS-A` Mammal, inherits properties of Mammal). Default inheritance is common (overridden by specific info).
  - **Spreading Activation:** Activating nodes and following links to find relationships or related concepts.
- **Semantics Issue:** Major weakness - syntax is simple, but the *meaning* (semantics) of nodes and links is often not formally defined, leading to ambiguity.
- **Advantages:** Intuitive visualization, good for hierarchical knowledge, network structure acts as an index for quick inference.
- **Disadvantages:** Lack of well-defined semantics (primary issue), difficult to represent negation and disjunction, choosing good primitives is hard.

6 Ontology

- **Definition:** A formal, explicit specification of a shared conceptualization. Essentially, a structured dictionary/vocabulary defining concepts, properties, and relationships within a specific domain.
- **Purpose:** Enable knowledge sharing and reuse between systems and people. Formalize domain knowledge.
- **Structure:** Defines entities/concepts, attributes, relations, constraints, often hierarchically (like SNs or Frames, but usually with more formal semantics).
- **Building Process:** Define scope/purpose -¿ Consider reuse -¿ List concepts -¿ Define classification/hierarchy -¿ Define attributes -¿ Define constraints/relations -¿ Add instances -¿ Check for anomalies/contradictions.
- **Examples:** `CYC` (commonsense), `WordNet` (lexical), `UMLS` (medical), `SUMO` (general upper ontology).
- **Key Idea:** Provides a common ground and formal structure for knowledge, often leveraging techniques from Frames and Semantic Networks but aiming for stricter semantic definitions (using formal languages like OWL).

Chapter 8: Machine Learning

1 Fundamental Concepts of Machine Learning (ML)

1.1 Definition of Machine Learning

- ML is a subfield of AI focused on building computer systems that **automatically improve with experience**.
- **Learning Problem Definition (Tom Mitchell):** A machine learns if its **performance P** at a specific **task T** improves with **experience E**. This is represented by the triple (T, P, E).

- **T (Task):** The problem the system is trying to solve (e.g., classifying emails, recognizing words).
- **P (Performance):** A metric to evaluate how well the system performs the task (e.g., % of correctly classified emails).
- **E (Experience):** The data the system uses to learn (e.g., a set of sample emails with labels).

1.2 The Learning Process

1. **Goal:** Learn an unknown target function $y^* : x \mapsto y$.
   - **x:** An observation, data instance, or past experience.
   - **y:** A prediction, new knowledge, or new experience.
2. **Learning:** Learn from a set of training examples (a **training set**).
3. **Outcome:** Produce a model/function $y = f(x)$ that approximates the target function y^*. This model is used for **prediction** or **inference** on new, future observations.

1.3 ML Workflow

- **Basic Process:**
  1. Split the full **Dataset** into a **Training Set** and a **Test Set**.
  2. Use the **Training Set** to **train** the system/model.
  3. Use the **Test Set** to **test** (evaluate) the performance of the trained system.
- **Careful Process (more robust):**
  1. Split the Dataset into **Training**, **Validation**, and **Test Sets**.
  2. **Training Phase:** Train the model on the Training Set.
  3. **Optimization Phase:** Use the **Validation Set** to tune system parameters (hyperparameters) and prevent overfitting.
  4. **Testing Phase:** Perform a final, unbiased evaluation of the model on the **Test Set**.

1.4 Key Challenges

- **Generalization:** The ultimate goal is for the model to perform well on new, unseen data, not just the data it was trained on.
- **Overfitting:** A critical problem where the model learns the training data too well (including its noise and quirks), resulting in high accuracy on the training set but low accuracy on the validation and test sets.

2 Types of Learning Problems

2.1 Supervised Learning

- **Goal:** Learn the function y^* from a labeled training set $\{(x_1, y_1), (x_2, y_2), \dots\}$.
- **Types:**
  - **Classification:** The output y belongs to a finite set of discrete categories or labels (e.g., {spam, normal}).
  - **Regression:** The output y is a continuous, real-valued number (e.g., predicting stock prices).

2.2 Unsupervised Learning

- **Goal:** Find hidden patterns or intrinsic structures in an unlabeled training set $\{x_1, x_2, \dots\}$.
- **Common Tasks:**
  - **Clustering:** Grouping similar data points together.
  - **Trend Detection:** Discovering trends and patterns in user data.

3 K-Nearest Neighbors (k-NN) Algorithm

A simple, non-parametric method for both classification and regression. It is also known as lazy learning or instance-based learning.

3.1 Main Ideas

- **Learning Phase:** Simply stores all the training data. There is no model-building step.
- **Prediction Phase:** For a new instance z, find the k most similar instances (neighbors) in the training data and use their labels to predict the label for z.

3.2 Algorithm Steps

1. For a new instance z, calculate the **distance/similarity** to every instance x in the training data D.
2. Identify the k instances in D that are closest to z (the nearest neighbors).
3. **For Classification:** Assign z the class label that is most frequent among its k nearest neighbors (majority vote).
4. **For Regression:** Assign z the average of the values of its k nearest neighbors.

3.3 Formulas

3.3.1 k-NN for Regression

$$y_z = \frac{1}{k} \sum_{x \in NB(z)} y_x$$

where $NB(z)$ is the set of k nearest neighbors of z.

3.3.2 Distance/Similarity Measures

- **Minkowski Distance (p-norm):**

$$d(x, z) = \left(\sum_{i=1}^{n} |x_i - z_i|^p \right)^{1/p}$$

- **Manhattan Distance (p=1):**

$$d(x, z) = \sum_{i=1}^{n} |x_i - z_i|$$

- **Euclidean Distance (p=2):**

$$d(x, z) = \sqrt{\sum_{i=1}^{n} (x_i - z_i)^2}$$

- **Chebyshev Distance (p=∞):**

$$d(x, z) = \max_i |x_i - z_i|$$

- **Hamming Distance (for binary inputs):** The number of positions at which the corresponding symbols are different.

$$d(x, z) = \sum_{i=1}^{n} \text{Difference}(x_i, z_i) \quad \text{where Difference}(a, b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases}$$

4 Probabilistic Learning & Naïve Bayes

4.1 Bayes' Theorem

This theorem is the foundation for probabilistic classification.

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- **P(h—D) (Posterior Probability):** Probability of hypothesis h being true given the data D.
- **P(D—h) (Likelihood):** Probability of observing data D if hypothesis h were true.
- **P(h) (Prior Probability):** Initial probability of hypothesis h being true.
- **P(D) (Evidence):** Probability of observing the data D.

4.2 MAP and MLE

- **Maximum a Posteriori (MAP) Hypothesis:** Find the hypothesis that is most probable given the data. Since $P(D)$ is constant, this is equivalent to maximizing the numerator.

$$h_{MAP} = \arg\max_{h \in H} P(D|h)P(h)$$

- **Maximum Likelihood Estimation (MLE):** A simplification of MAP that assumes all hypotheses have the same prior probability ($P(h)$ is uniform). The goal is to find the hypothesis that maximizes the likelihood of the data.

$$h_{ML} = \arg\max_{h \in H} P(D|h)$$

4.3 Naïve Bayes Classifier

- **Goal:** For a new instance z with attributes $(z_1, z_2, \ldots, z_n)$, find the most probable class c.
- **"Naïve" Assumption (Conditional Independence):** It assumes that all attributes are independent of each other given the class label. This is a strong, "naïve" assumption, but works surprisingly well in practice.

$$P(z_1, \ldots, z_n|c_i) = \prod_{j=1}^{n} P(z_j|c_i)$$

- **Naïve Bayes Classifier Formula:** Combines the MAP rule with the naïve assumption.

$$c_{NB} = \arg\max_{c_i \in C} P(c_i) \prod_{j=1}^{n} P(z_j|c_i)$$

4.4 Practical Issues and Solutions

1. **Zero-Frequency Problem:** If an attribute value x_j never appears with a class c_i in the training set, $P(x_j|c_i) = 0$, causing the entire product to become zero.
   - **Solution (Additive/Laplace Smoothing):** Add a small value m (often 1) to the count for each attribute-class combination.

$$P(x_j|c_i) = \frac{n(c_i, x_j) + mp}{n(c_i) + m}$$

   - $n(c_i)$: number of training examples in class c_i.
   - $n(c_i, x_j)$: number of examples in class c_i with attribute value x_j.
   - m: equivalent sample size (a smoothing parameter).
   - p: prior probability for the attribute (e.g., $1/k$ if attribute has k values).

2. **Numerical Underflow:** Multiplying many small probabilities can result in a floating-point value of zero.

    - **Solution (Log-Probabilities):** Perform calculations using the sum of logarithms, which is numerically more stable.

$$c_{NB} = \arg\max_{c_i \in C} \left[\log(P(c_i)) + \sum_{j=1}^{n} \log(P(z_j|c_i)) \right]$$